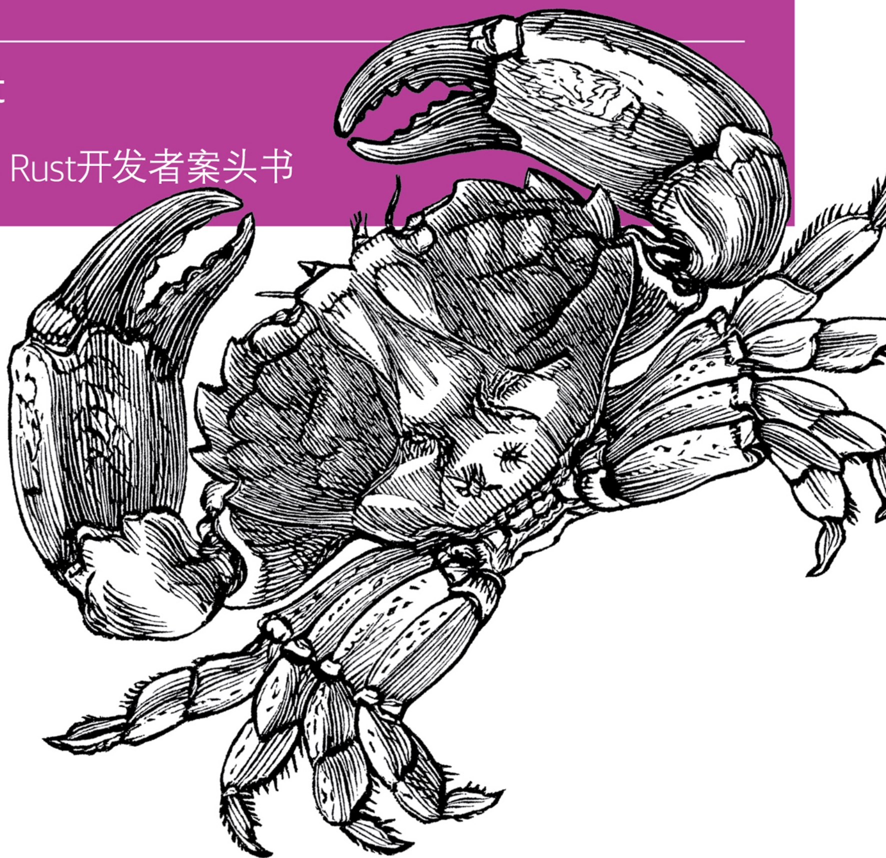
O'REILLY®

TURING 图灵程序设计丛书

Rust程序设计

Programming Rust

理论与实战相结合，Rust开发者案头书



[美] 吉姆·布兰迪 贾森·奥伦多夫 著

李松峰 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

李松峰

360前端开发资深专家、前端TC委员、W3C AC代表，任职于“奇舞团”，也是360 Web字体服务“奇字库”作者。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。


图灵程序设计丛书

Rust 程序设计

Programming Rust:
Fast, Safe Systems Development

[美] 吉姆·布兰迪　贾森·奥伦多夫 著
李松峰 译

Beijing • Boston • Farnham • Sebastopol • Tokyo　O'REILLY®

O'Reilly Media, Inc. 授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　京

图书在版编目（ＣＩＰ）数据

Rust程序设计 / （美）吉姆•布兰迪（Jim Blandy），
（美）贾森•奥伦多夫（Jason Orendorff）著 ; 李松峰 译.
-- 北京 : 人民邮电出版社, 2020.9
（图灵程序设计丛书）
ISBN 978-7-115-54649-4

Ⅰ. ①R… Ⅱ. ①吉… ②贾… ③李… Ⅲ. ①程序语
言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2020)第148740号

内 容 提 要

本书由两位经验丰富的系统程序员撰写，介绍了一种具有 C 和 C++ 性能，同时安全且支持并发的新型系统编程语言 Rust，解释了 Rust 如何在性能和安全性之间架起桥梁，以及我们如何用好这门语言。书中主要内容包括：Rust 的基本数据类型，关于所有权、引用等概念，表达式、错误处理、包和模块、结构体、枚举与模式等基础知识，Rust 语言的特型与泛型，闭包，迭代器，集合，等等。

本书适合系统程序员以及有其他编程语言经验的开发者阅读。

◆ 著　　　　[美] 吉姆 • 布兰迪　贾森 • 奥伦多夫
　译　　　　李松峰
　责任编辑　张海艳
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　https://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：800 × 1000　1/16
　印张：31
　字数：733千字　　　　　2020年 9 月第 1 版
　印数：1 – 3 000册　　　2020年 9 月北京第 1 次印刷

著作权合同登记号　图字：01-2020-4162 号

定价：139.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东市监广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

Rust 是一门系统编程语言。

如今，这个定位需要稍微解释一下，因为现在大多数以编程为业的工程师并不熟悉系统编程。然而只有理解系统编程才能更好地认识本书的意义。

你合上了自己的笔记本计算机。你的操作系统检测到这个动作，挂起所有运行的程序，关闭显示器，让计算机进入休眠状态。过了一会儿，你打开笔记本计算机，屏幕和其他应用便随之启动，每个程序都恢复到了之前的状态。我们认为这是理所当然的，但系统程序员为这一切编写了很多代码。

所谓系统编程，指的是编写：

- 操作系统
- 各种设备驱动
- 文件系统
- 数据库
- 运行在廉价设备或必须极端可靠设备上的代码
- 加解密程序
- 媒体编解码器（读写音频、视频和图片文件的软件）
- 媒体处理器（如语音识别或图片编辑软件）
- 内存管理程序（如实现垃圾收集器）
- 文本渲染程序（将文本和字体转换为像素）
- 高级编程语言（如 JavaScript 或 Python）
- 网络程序
- 虚拟化及软件容器
- 科学模拟程序
- 游戏

简言之，系统编程是一种**资源受限**的编程。这种编程需要对每个字节和每个 CPU 时钟周期精打细算。

支持一个简单程序运行所涉及的系统代码的数量是惊人的。

本书不会教你系统编程。事实上，本书会介绍很多内存管理的细节，如果没有一定的系统编程经验，这些乍一看有点过于深奥。但是，如果你是一名资深系统程序员，就会发现 Rust 有点不可思议。这门语言解决了困扰整个行业几十年、人所共知的重要问题。

读者对象

如果你就是一名系统程序员，而且在寻找 C++ 的替代品，那应该看看本书。如果你是一名有其他编程语言经验的开发者，无论是 C#、Java、Python、JavaScript，抑或别的语言，那也应该看看本书。

不过，你不仅仅需要学习 Rust。为了更好地理解这门语言，你也需要有一些系统编程的经验。建议你在阅读本书的同时，也用 Rust 写几个系统编程的项目。利用 Rust 的速度、并发和安全，构建一些从来没有构建过的应用。前面列出的那个清单应该能给你一点启发。

写作初衷

我们想写一本自己当初学习 Rust 时希望看的书。我们的目标是首先把 Rust 重要的新概念摆出来，直面问题，然后再把它们清晰、深入地讲清楚，通过反复尝试降低学习难度。

本书内容

本书前两章介绍 Rust 并提供了一个简单教程，第 3 章讲解基本的数据类型，第 4 章和第 5 章解释所有权和引用的核心概念。建议大家按顺序阅读前 5 章。

第 6 章到第 10 章讨论这门语言的基础，包括表达式（第 6 章）、错误处理（第 7 章）、包和模块（第 8 章）、结构体（第 9 章），以及枚举与模式（第 10 章）。这些章节可以跳着读，但一定不要跳过第 7 章。相信我们。

第 11 章介绍特型与泛型，这也是你需要知道的最后两个主要概念。特型类似于 Java 或 C# 中的接口。它们也是 Rust 支持的把你的类型集成到这门语言本身的主要方式。第 12 章展示如何通过特型实现操作符重载，第 13 章介绍更多的实用特型。

理解特型和泛型之后就可以阅读本书剩下的内容了。闭包和迭代器是两个不容错过的强大工具，第 14 章和第 15 章分别对它们进行了介绍。剩下的所有章节可以按任意顺序阅读，或者根据需要来研究。这些章节涵盖了这门语言的其他部分：集合（第 16 章）、字符串与文本（第 17 章）、输入和输出（第 18 章）、并发（第 19 章）、宏（第 20 章）以及不安全代码（第 21 章）。

排版约定

本书使用以下排版约定。

- **黑体**

  表示新术语和重点强调的内容。

- 等宽字体（`constant width`）

  用于程序清单，以及在段落内表示程序元素，比如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

- 加粗等宽字体（**`constant width bold`**）

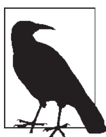
  表示应该由用户输入的命令或其他文本。

- 斜体等宽字体（*`constant width italic`*）

  表示应该使用用户提供的值或根据上下文确定的值来替换的文本。



此图标代表提示或建议。



此图标代表一般性说明。



此图标代表警告或提醒。

使用代码示例

书中示例的源代码请到图灵社区本书主页 http://ituring.cn/book/2101“随书下载”处下载。

本书是帮助你完成工作的。一般来说，如果代码示例是本书提供的，你可以在自己的程序或文档中使用。除非大量复制本书代码，否则你不需要联系我们获取授权。例如，使用本书中的几段代码编写一个程序不需要获取授权。销售或者发布 O’Reilly 图书中代码的光盘，则需要获取授权。引用本书以及示例代码来回答问题不需要获取授权。将本书中的大量示例代码整合到你的产品文档中，则需要获取授权。

在使用代码时，我们希望能标明出处，但并不强求。出处一般包括书名、作者、出版商和 ISBN。例如，“*Programming Rust* by Jim Blandy and Jason Orendorff (O’Reilly). Copyright 2018 Jim Blandy and Jason Orendorff, 978-1-491-92728-1”。

如果还有关于使用代码的未尽事宜，可以随时与我们联系：permissions@oreilly.com。

O'Reilly在线学习平台（O'Reilly Online Learning）

O'REILLY®

近 40 年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 http://oreilly.com。

联系我们

与本书有关的评论和问题，请发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

请访问 http://oreilly.com，到本书页面查看相关勘误。[1]

与本书有关的评论或技术问题，请发送电子邮件至 bookquestions@oreilly.com。

要了解更多我们出版的图书、课程、会议和新闻，请访问 http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

请在 YouTube 上关注我们：http://www.youtube.com/oreillymedia。

致谢

本书很大程度上得益于我们官方技术审稿人的关注，感谢 Brian Anderson、Matt Brubeck、J. David Eisenberg 和 Jack Moffitt。

还有很多非官方审稿人阅读了早期的草稿并提供了有价值的反馈。感谢 Eddy Bruel、Nick Fitzgerald、Michael Kelly、Jeffrey Lim、Jakob Olesen、Gian-Carlo Pascutto、Larry Rabinowitz、Jaroslav Šnajdr 和 Joe Walker 为本书提出的中肯意见。Jeff Walden 和 Nicolas Pierron 特别慷慨地花时间几乎审校了全书。与任何编程冒险一样，一本编程书也会因为高质量的 bug 报告而茁壮成长。谢谢你们。

注 1：本书中文版勘误请到 http://ituring.cn/book/2101 查看和提交。——编者注

Mozilla 对我们在这个项目上的工作极为包容，即使这件事并非我们工作职责所在，而且还会占用一部分工作时间。非常感谢 Dave Camp、Naveed Ihsanullah、Tom Tromey 和 Joe Walker 这几位领导的支持。他们都在为 Mozilla 的长远着想，我们希望这份成果能够达到他们的期望。

我们还想表达对 O'Reilly 每一位编辑的感激之情，是他们让这个项目得以开花结果，尤其要感激 Brian MacDonald 和 Jeff Bleiel。

最重要的是，衷心感谢我们的妻子和孩子，感谢他们坚定不移的爱、热情和耐心。

电子书

扫描如下二维码，即可购买本书中文版电子版。



第1章

为什么是Rust

> 在某些场景，比如 Rust 的应用场景下，速度是竞品的 10 倍，哪怕只是两倍都是关系到生死存亡的大问题。速度决定了这套系统在市场上的命运，跟硬件市场丝毫不差。
>
> ——Graydon Hoare

> 如今，所有计算机都是并行的……并行编程**就是**编程。
>
> ——*Structured Parallel Programming*，Michael McCool 等

> TrueType 解析器的缺陷被攻击者利用，以达到监视目的；所有软件都涉及安全。
>
> ——Andy Wingo

从开始使用高级语言编写操作系统至今的 50 多年来，系统编程语言已经取得了长足的进步。然而，有两个问题至今仍无法破解。

- 要写出安全的代码并不容易。想用 C 和 C++ 恰当地管理好内存特别困难。几十年来，用户已经饱受安全漏洞的侵害，至少可以追溯到 1988 年的 Morris 蠕虫病毒。
- 编写多线程代码非常困难，而多线程又是充分利用现代计算机能力的唯一方式。即便是经验丰富的程序员，在应对与线程有关的代码时也必须多加留心，因为并发会导致各式各样的新 bug，还会让普通的 bug 难以复现。

下面介绍一种具有 C 和 C++ 性能，同时安全且支持并发的语言——Rust。

Rust 是由 Mozilla 和社区贡献者共同开发的一种新的系统编程语言。与 C 和 C++ 类似，Rust 为开发者使用内存提供了完善的控制机制，在语言的原始操作与运行它的机器的操作之间维护着一种紧密的关系，让开发者能够预测自己代码的运行成本。Rust 承载着

C++ 之父 Bjarne Stroustrup 在他的论文“Abstraction and the C++ Machine Model”中明确提出的理想：

> 总的来说，C++ 实现遵循零开销原则：不用的，不必为之付出代价；要用的，也不会有代码比它更好。

在此基础之上，Rust 又追加了自己内存安全和可靠并发的目标。

Rust 实现上述所有承诺的关键在于**所有权**（ownership）、**转移**（move）和**借用**（borrow）机制造就的新型系统，而编译时检查和认真的设计又成就了 Rust 灵活的静态类型系统。所有权机制为每个值规划了清晰的生命期，从而让核心语言不再需要垃圾收集，同时还为管理套接口（socket）和文件勾柄（handle）[1] 等资源提供了可靠而又灵活的接口。转移把值从一个所有者转移给另一个所有者，而借用让代码可以临时使用某个值，同时又不影响其所有权。考虑到很多程序员之前从未碰到过此类特性的这种形式，第 4 章和第 5 章将对它们进行详尽的介绍。

同样的所有权规则也是 Rust 值得依赖的并发模型的基础。说到互斥量（mutex）与其所要保护数据的关系，大多数语言是靠注释解决问题的。而 Rust 通过编译时检查可以发现访问被锁住的互斥量的问题。大多数语言只会告诫开发者要确保不访问已经交给其他线程的数据，Rust 却能通过检查保证你没有那么做。Rust 能够在编译时防止数据争用。

Rust 并非真正的面向对象语言，它只是具有一些面向对象的特征而已。Rust 也不是函数式语言，虽然它可以像函数式语言那样让计算结果更容易推断。Rust 在某种程度上类似于 C 和 C++，但 C 和 C++ 的很多惯用语法不能照搬过来在 Rust 中使用，所以典型的 Rust 代码说到底还是不像 C 或 C++ 代码。关于 Rust 到底是哪种语言，最好还是等你熟悉它之后，自己来下结论吧。

为了通过真正的项目获得关于设计的反馈，Mozilla 用 Rust 开发了 Servo，这是一个新的 Web 浏览器引擎。Servo 的需求与 Rust 的目标完美匹配：浏览器必须高性能，还要能安全地处理不受信的数据。Servo 利用 Rust 的安全并发最大限度地挖掘机器潜力，在某些任务上实现了 C 或 C++ 不可能实现的并行处理。Servo 与 Rust 一直并肩成长，Servo 不断应用 Rust 的最新语言特性，Rust 也基于 Servo 开发者的反馈不断改进。

类型安全

Rust 是类型安全的语言，那么“类型安全”指的是什么？安全听起来不错，但要从哪里做起呢？

以下是 C99，也就是 C 编程语言 1999 年标准中对**未定义行为**的定义：

> **未定义行为**
>
> 由于使用不可移植或错误的程序构造，或者使用错误的数据导致的行为，本国际标准对此不作要求。

注 1：大多数人习惯于将 handle 翻译为“句柄”，但“勾柄”才是正确的。——译者注

来看下面的 C 程序：

```
int main(int argc, char **argv) {
  unsigned long a[1];
  a[3] = 0x7ffff7b36cebUL;
  return 0;
}
```

根据 C99，因为这段程序访问的元素超出了数组 a 的边界，所以它的行为是未定义的。换句话说，执行这段代码可以出现任何结果。当我们在 Jim 的笔记本计算机上运行这段程序时，看到了以下输出：

```
undef: Error: .netrc file is readable by others.
undef: Remove password or make file unreadable by others.
```

然后程序就崩溃了。Jim 的笔记本计算机上根本就没有一个叫 .netrc 的文件。如果你在自己的计算机上运行这段代码，很可能结果又不一样。

C 编译器为这个 main 函数生成的机器码恰好把数组 a 保存到返回地址前面 3 个字的位置，因此把 0x7ffff7b36cebUL 保存到 a[3]，会把 main 的返回地址改为指向 C 标准库中一段代码的中间，该代码会从某人的 .netrc 文件中读取密码。在 main 返回时，执行并没有恢复到 main 的调用者，而是转到了库中以下几行代码的机器码：

```
warnx(_("Error: .netrc file is readable by others."));
warnx(_("Remove password or make file unreadable by others."));
            goto bad;
```

C 编译器允许数组引用影响后续 return 语句的行为是完全符合标准的。未定义操作并非只产生意想不到的结果，事实上这种情况下程序**无论做任何事情**都是被允许的。

为了生成更快的代码，C99 授予编译器全权。这个标准没有让编译器负责检测和处理可疑的行为（比如数组越界），而是让程序员负责保证这种情况永远不会发生。

从经验来看，人类在这方面并不擅长。在犹他大学读书时，研究员李朋（Peng Li）修改了 C 和 C++ 编译器，让它们编译后的程序可以在执行某种形式的未定义行为时发送报告。他发现几乎所有程序都会发送报告，包括那些高标准严要求的备受推崇的项目。实践中，未定义行为经常会导致可被利用的安全漏洞。Morris 蠕虫病毒就是利用前面代码的原理并经过精心改造，将自己复制到不同机器的。而基于同样原理的漏洞利用在今天仍然广泛存在。

基于这个例子，可以定义几个术语。如果将一个程序写得不可能在执行时导致未定义行为，那么就称这个程序为**定义良好的**（well defined）。如果一种语言的安全检查可以保证所有程序都定义良好，那么就称这种语言是**类型安全的**。

如果足够用心，用 C 或 C++ 应该也能写出定义良好的程序，但 C 和 C++ 不是类型安全的：前面的程序中没有类型错误，但出现了未定义行为。相对而言，Python 是类型安全的。Python 乐意花处理器时间来检查和处理数组索引越界的操作，方式也比 C 更友好：

```
>>> a = [0]
>>> a[3] = 0x7ffff7b36ceb
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

Python 抛出了异常，这不是未定义行为：Python 文档指出，`a[3]` 这种赋值应该抛出 `IndexError` 异常（我们也看到了）。当然，`ctypes` 之类提供对机器无约束访问的模块可能会在 Python 代码中引入未定义行为，但其核心语言本身还是类型安全的。Java、JavaScript、Ruby 和 Haskell 在这方面都类似。

注意，类型安全与一门语言是在编译时还是在运行时检查类型无关。C 在编译时检查，但它不是类型安全的；Python 在运行时检查，但它是类型安全的。

说起来还真有点尴尬，占有统治地位的系统编程语言 C 和 C++ 都不是类型安全的，大多数其他流行的语言则是类型安全的。考虑到设计 C 和 C++ 的初衷就是用它们去实现系统的基础部分、实现可靠的安全隔离，以及操作不可信数据，类型安全对它们而言好像恰恰是最有价值的特性才对。

Rust 要解决的正是这个沉淀了几十年的老问题：它既是类型安全的，又是一种系统编程语言。Rust 的设计初衷也是用于实现那些要求高性能和对资源精细控制的基础系统层，同时还能基于类型安全提供最基本的可预测性。本书后面的章节将详细介绍 Rust 是如何做到这个统一的。

对多线程编程而言，Rust 特定形式的类型安全有着令人意想不到的影响。众所周知，正确实现并发在 C 和 C++ 中非常困难。开发者通常仅在单线程实在无法满足性能要求时才会考虑并发。但 Rust 可以通过编译时检查保证并发不会发生数据争用，并会捕获任何对互斥量或者其他同步原语（synchronization primitive）的错误使用。使用 Rust 编写并发程序，你再也不用担心自己的代码只有经验非常丰富的程序员才能看懂了。

Rust 的安全规则也有一个“逃生阀”，用于必须使用原始指针的情形。这种情况下你写的是**不安全代码**，虽然绝大多数 Rust 程序用不到，但第 21 章也会介绍如何使用它以及它在整个 Rust 安全蓝图中的位置。

与其他静态类型的语言相似，Rust 的类型除了可以防止未定义行为，还有很多优点。经验丰富的 Rust 程序员利用类型不仅可以保证安全地使用数据，也可以保证有意义地使用数据，即与应用的意图保持一致。特别地，第 11 章讨论的 Rust 的特型（trait）和泛型（generic），为描述一组类型的共性，乃至进一步利用这些共性提供了简洁、灵活且高效的手段。

本书的宗旨不仅是教会你如何用 Rust 编写程序，更重要的是要教会你如何利用这门语言写出安全又得体的程序，同时还能够预测程序的执行。在我们看来，Rust 应该是系统编程发展史上的一个巨大进步，本书就是要帮你学会使用它。

第2章 Rust初体验

一个人的经验完全建构在他的语言之上。

——Henri Delacroix（法国心理学家）

本章将通过几个小程序来展示 Rust 的语法、类型及语义，看看它们是如何支持安全、并发和高效代码的。我们会从下载、安装 Rust 开始，然后看一个数学相关的小例子，接下来尝试基于第三方库写一个 Web 服务器，最后再使用多线程加速绘制曼德布洛特（Mandelbrot）集合的过程。

2.1 下载和安装Rust

安装 Rust 的最佳方法是使用 `rustup`，也就是 Rust 安装器。打开 rustup.rs 网站，然后按照上面的指示操作即可。

此外，也可以打开 Rust 官方网站，点击 Install，在 Other installation methods（其他安装方法）中找到针对 Linux、macOS 和 Windows 的单独安装包。某些操作系统中也可能已经自带了 Rust。但还是推荐大家使用 `rustup` 来安装，因为它是一个管理 Rust 安装包的工具，类似 Ruby 的 RVM 或者 Node 的 NVM。这样在遇到 Rust 有新版本发布时，只要运行 `rustup update` 即可升级到新版本。

不管怎样，安装完成后，都应该可以在命令行中使用 3 个新命令：

```
$ cargo --version
cargo 1.34.0 (6789d8a0a 2019-04-01)
$ rustc --version
rustc 1.34.1 (fc50f328b 2019-04-24)
$ rustdoc --version
rustdoc 1.34.1 (fc50f328b 2019-04-24)
$
```

这里的 `$` 是命令提示符，在 Windows 上它应该是 `C:\>` 或类似的东西。上面的代码中运行了我们安装的 3 个命令，分别输出了它们的版本号。下面分别介绍一下这几个命令。

- `cargo` 是 Rust 的编译管理器、包管理器以及通用工具。可以使用 Cargo 来创建新项目、构建和运行程序，以及管理代码所依赖的外部库。
- `rustc` 是 Rust 编译器。通常是通过 Cargo 来调用编译器，但有时候也需要直接调用它。
- `rustdoc` 是 Rust 文档工具。如果在代码注释中以适当格式写了文档，那么 `rustdoc` 可以基于它们生成格式化的 HTML。与 `rustc` 类似，通常也让 Cargo 来帮助运行 `rustdoc`。

为方便起见，Cargo 也可以帮我们创建新的 Rust 包，并适当写入一些标准的元数据：

```
$ cargo new --bin hello
     Created binary (application) `hello` project
```

这个命令会创建一个新的包目录 hello，其中 `--bin` 标记告诉 Cargo 将其作为一个可执行文件，而不是一个库。下面看看这个包的顶级目录下都有什么：

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x.  4 jimb jimb 4096 Sep 22 21:09 .
drwx------. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x.  6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--.  1 jimb jimb    7 Sep 22 21:09 .gitignore
-rw-rw-r--.  1 jimb jimb   88 Sep 22 21:09 Cargo.toml
drwxrwxr-x.  2 jimb jimb 4096 Sep 22 21:09 src
$
```

可以看到，Cargo 创建了一个名为 Cargo.toml 的文件，用于保存这个包的元数据。此时文件内容不多：

```
[package]
name = "hello"
version = "0.1.0"
authors = ["You <you@example.com"]
edition = "2018"

[dependencies]
```

如果程序依赖其他库，那么可以把它们列在这个文件里，Cargo 将负责下载、构建和更新这些库。关于 Cargo.toml 的配置，第 8 章将详细介绍。

Cargo 刚刚创建了一个自带 Git 版本控制系统的包，.git 子目录中包含着元数据，还有一个 .gitignore 文件。如果不需要 Cargo 这么做，那么可以在命令行里加上 `--cvs none` 标记。

`src` 子目录中包含了真正的 Rust 代码：

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

看起来好像 Cargo 已经替我们写好程序了。main.rs 文件中包含以下内容：

```
fn main() {
    println!("Hello, world!");
}
```

看见了吗，使用 Rust 你甚至不用自己写“Hello, world!”程序。应该说，这就是一个新 Rust 程序的样板：两个文件，总共 10 行代码。

在这个包的任何目录调用 `cargo run` 命令都可以构建并运行这个程序：

```
$ cargo run
   Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27 secs
     Running `/home/jimb/rust/hello/target/debug/hello`
Hello, world!
$
```

这里，Cargo 调用了 Rust 编译器 `rustc`，然后又运行了生成的可执行文件。Cargo 把可执行文件放到了包顶部的 target 子目录中：

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb    198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb     68 Sep 22 21:37 incremental
drwxrwxr-x. 2 jimb jimb   4096 Sep 22 21:37 native
$ ../target/debug/hello
Hello, world!
$
```

最后，可以让 Cargo 清理生成的文件：

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
$
```

2.2　一个简单的函数

Rust 的语法有意模仿了现有语言。如果你熟悉 C、C++、Java 或 JavaScript，那自然就可以看懂 Rust 程序的结构。下面是一个用于计算两个整数的最大公约数的函数，其使用的是欧几里得算法：

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
            let t = m;
            m = n;
            n = t;
        }
```

```
        m = m % n;
    }
    n
}
```

关键字 fn（发音为 [fʌn]）定义了一个名为 gcd（greatest common divisor，最大公约数）的函数，其接收两个参数 n 和 m，类型均为 u64，即无符号（unsigned）64 位（bit）整数。符号 -> 后面是返回值类型，这里同样也返回 u64 值。4 个空格的缩进是 Rust 风格。

Rust 的机器整数类型名反映了它们的大小及有无符号，比如 i32 表示有符号 32 位整数，u8 表示无符号 8 位整数（用于“字节”值）。isize 和 usize 分别表示指针大小的有符号和无符号整数，在 32 位平台是 32 位，在 64 位平台则为 64 位。Rust 还有两个浮点类型 f32 和 f64，分别是 IEEE 单精度和双精度浮点类型，类似 C 和 C++ 的 float 和 double。

默认情况下，一个变量被初始化之后，它的值就不能变了。但是，像前面那样在参数 n 和 m 前加上 mut（发音为 [mjut]，是 mutable 的简写）关键字，则表示可以在函数体内给它们赋值。实践中，大多数变量不会被赋值，此时加上 mut 关键字可以给阅读代码的人一个提醒。

函数体的第一行代码调用了 assert! 宏，用于验证参数的值都不等于 0。! 符号表示这是一个宏调用，不是函数调用。与 C 和 C++ 中的 assert 宏类似，Rust 中的 assert! 宏会检查自己的参数是不是 true，如果不是，则终止程序并输出相关信息，包含检查失败的代码在源代码中的位置。这种突然的终止在 Rust 中叫**诧异**（panic）。与 C 和 C++ 中的断言可以跳过不同，Rust 不管程序是如何编译的都会检查断言。不过也有一个 debug_assert! 宏，它会在程序为速度而编译时被跳过。

我们的函数的核心是一个 while 循环，其中又包含一个 if 语句和一个赋值。与 C 和 C++ 不同，Rust 不需要用圆括号来括住条件表达式，但需要用花括号括住条件满足后要执行的语句。

let 语句用于声明一个局部变量，比如函数中的 t。这里不用明确写出 t 的类型，因为 Rust 可以根据如何使用变量推断出来。在这个函数中，唯一适合 t 的类型是 u64，也就是 m 和 n 的类型。Rust 只在函数体内推断变量类型，函数参数和返回值则必须像前面一样明确写出类型。假如要写出 t 的类型，则可以这样写：

```
let t: u64 = m;
```

Rust 有 return 语句，但 gcd 函数中没有。如果函数体中最后一行代码是一个表达式，且表达式末尾**没有**分号，那这个表达式的值就是函数的返回值。实际上，用花括号括起来的任何代码块都可以看作一个表达式。比如，下面这个代码块就是一个打印一条消息之后又返回 x.cos() 值的表达式：

```
{
    println!("evaluating cos x");
    x.cos()
}
```

这种在控制流“离开函数末尾”时生成函数值的方式是 Rust 特有的。return 语句一般只用于在函数的中间提前返回。

2.3 编写和运行单元测试

Rust 语言本身内置了简单测试机制。要测试 `gcd` 函数，可以这样写：

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                   3 * 7 * 11 * 13 * 19),
               3 * 11);
}
```

这里定义了一个名为 `test_gcd` 的函数，它会调用 `gcd` 并检查其是否返回了正确的值。函数定义上方的 `#[test]` 表示 `test_gcd` 是一个测试函数，在常规编译中会被跳过，但在通过 `cargo test` 命令运行程序时会包含并自动调用。假设我们修改了本章开始时创建的 hello 包，加入了 `gcd` 和 `test_gcd` 的代码。如果当前目录是这个包中的任意目录，则可以像下面这样运行测试：

```
$ cargo test
   Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.35 secs
     Running /home/jimb/rust/hello/target/debug/deps/hello-2375a82d9e9673d7

running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

$
```

测试函数可以写在源代码中的任何地方，只要紧跟着它要测试的代码即可。这样，`cargo test` 会自动把它们收集起来并全部运行。

`#[test]` 标记是**属性**（attribute）的一个例子。属性是一种开放式标记机制，用于给函数或其他声明添加补充说明，就像 C++ 和 C# 中的属性和 Java 中的注解一样。属性可以用于控制编译器报警和代码风格检查、有条件地包含代码（比如 C 和 C++ 中的 `#ifdef`）、告诉 Rust 如何与其他语言的代码互操作，等等。本书后面还会介绍更多属性的例子。

2.4 处理命令行参数

如果想通过命令行参数接收一系列数值，然后打印出它们的最大公约数，那么可以将 `main` 函数改写成下面这样：

```
use std::io::Write;
use std::str::FromStr;

fn main() {
    let mut numbers = Vec::new();
```

```
    for arg in std::env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }

    if numbers.len() == 0 {
        writeln!(std::io::stderr(), "Usage: gcd NUMBER ...").unwrap();
        std::process::exit(1);
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    println!("The greatest common divisor of {:?} is {}",
             numbers, d);
}
```

代码不少，我们拆开来看：

```
use std::io::Write;
use std::str::FromStr;
```

`use` 声明向当前作用域引入了两个**特型**（trait）：`Write` 和 `FromStr`。第 11 章将详细介绍特型，在此只要简单地把它们理解为某种类型可以实现的一组方法就行了。虽然程序中没有用到 `Write` 或 `FromStr` 这两个名字，但用到了它们的方法，为此相关的特型必须出现在作用域中。对当前的例子而言，有下面两种情况。

- 任何实现 `Write` 特型的类型都有用于将格式化文本写入输出流的 `write_fmt` 方法。`std::io::Stderr` 类型实现了 `Write`，而我们要使用 `writeln!` 宏来打印错误消息，这个宏展开之后的代码要使用 `write_fmt` 方法。
- 任何实现 `FromStr` 特型的类型都有用于从字符串解析出该类型值的 `from_str` 方法。`u64` 类型实现了 `FromStr`，而我们会调用 `u64::from_str` 来解析命令行参数。

回到程序的 `main` 函数：

```
fn main() {
```

这个 `main` 函数不返回值，因此可以省略通常要写在参数列表后面的 `->` 和返回值类型声明。

```
let mut numbers = Vec::new();
```

在函数体内声明一个可修改局部变量 `numbers`，并将其值初始化为一个空向量（vector）。`Vec` 是 Rust 中可扩展的向量类型，类似 C++ 中的 `std::vector`、Python 中的列表或 JavaScript 中的数组。尽管向量是可扩展和收缩的，但要向向量末尾推入数值，Rust 还要求必须给变量加上 `mut` 标记。

`numbers` 的类型是 `Vec<u64>`，即一个包含 `u64` 值的向量。但跟以前一样，这里不需要明确写出来。Rust 会帮我们推断它的类型，一方面我们推到向量里的是 `u64` 值，另一方面我们传给 `gcd` 函数的是向量的元素，而 `gcd` 只接收 `u64` 值。

```
for arg in std::env::args().skip(1) {
```

接下来使用 `for` 循环处理命令行参数，依次将每个参数赋值给变量 `arg` 并运行循环体。

`std::env::args` 函数返回一个**迭代器**（iterator），而迭代器会按需生成每一个参数，并在没有更多参数时指出来。迭代器在 Rust 中极其常用，标准库中也包含多种迭代器，比如迭代向量元素的、迭代文件的每一行的、迭代收到的每条消息的，总之，可以通过循环处理的任何数据都有对应的迭代器。Rust 的迭代器效率很高，编译器通常可以把它们转换成跟手写循环一样的代码。第 15 章将讨论迭代器的原理和相应的例子。

除了与 `for` 循环配合使用，迭代器本身也提供了很多能直接使用的方法。比如，`std::env::args` 返回的迭代器的第一个值始终是当前运行的程序的名字。我们想跳过这个值，因此可以调用迭代器的 `skip` 方法，从而生成一个不包含第一个值的新迭代器。

```
numbers.push(u64::from_str(&arg)
             .expect("error parsing argument"));
```

接下来调用 `u64::from_str`，尝试从命令行参数 `arg` 中解析出一个无符号 64 位整数。注意，`u64::from_str` 并不是我们获取的某个 `u64` 值的方法，而是 `u64` 类型的方法，就跟 C++ 或 Java 中的静态方法类似。而且 `from_str` 函数也不直接返回一个 `u64` 值，而是返回一个表示解析成功或失败的 `Result` 类型的值。`Result` 类型的值有两种：

- `Ok(v)` 表示解析成功，`v` 为得到的值；
- `Err(e)` 表示解析失败，`e` 为包含错误信息的值。

所有涉及输入、输出或其他与操作系统交互的函数，都会返回 `Result` 类型的值。如果返回的是 `Ok` 值，则其中会包含操作成功的结果，可能是传输的字节数，也可能是打开的文件，等等；如果返回的是 `Err` 值，则其中会包含系统返回的错误码。与大多数现代编程语言不同，Rust 没有异常的概念：所有错误都使用 `Result` 或诧异来处理，第 7 章将详细介绍。

接下来，要使用 `Result` 的 `expect` 方法检查解析的结果。如果结果是某种 `Err(e)`，`expect` 就会输出 `e` 中包含的错误消息并立即退出程序。而如果结果是 `Ok(v)`，`expect` 则直接返回 `v`，最终这个值又会被推到数值向量的末尾。

```
if numbers.len() == 0 {
    writeln!(std::io::stderr(), "Usage: gcd NUMBER ...").unwrap();
    std::process::exit(1);
}
```

如果向量中根本没有数值，那么自然没办法计算最大公约数。因此接下来要检查向量中至少有一个元素，如果一个元素也没有则退出程序。退出程序之前，先调用 `writeln!` 宏向标准错误输出流（由 `std::io::stderr()` 返回）中写入错误消息。最后调用 `.unwrap()` 是检查输出错误消息这个操作本身没有失败的一种快捷方式；当然调用 `expect` 也可以检查，只是没必要那么小题大做。

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

紧接着又是一个循环，循环中使用变量 d 作为媒介，保存每次循环计算得到的最大公约数。跟前面一样，必须将 d 标记为可修改才可以在循环中给它赋值。

关于这个 for 循环，有两个地方比较奇怪。首先，我们写的是 for m in &numbers[1..]，这里的 & 操作符是干什么用的？其次，我们又写了 gcd(d, *m)，那么 *m 中的 * 又作何解？其实这两个细节是相辅相成的。

在此之前，我们的代码操作的都是像整数这种占据固定大小内存空间的简单值。而现在要迭代一个向量，其大小并不确定，有可能会非常大。Rust 对待这种值非常慎重，它希望程序员来控制内存用度，明确指出每个值存活多久，同时还要保证不需要时立即释放内存。

为此在迭代向量时要告诉 Rust，向量的**所有权**仍然属于 numbers，循环中仅仅是**借用**其元素而已。&numbers[1..] 中的 & 操作符表示从向量的第二个元素开始，借用每个元素的**引用**。for 循环迭代的是每个元素的引用，进而让 m 再去借用每个元素。*m 中的 * 操作符表示对 m **解引用**，即取得引用所指的值，也就是要传给 gcd 函数的第二个 u64 值。最终，因为还是 numbers 拥有向量，所以 Rust 会在 numbers 脱离 main 函数末尾的函数作用域时自动将向量释放。

Rust 设计的所有权和引用机制是其内存管理及安全并发的关键，具体细节第 4 章和第 5 章将讨论。要想得心应手地编写 Rust 代码，必须熟悉这些规则。但就初次体验而言，只要知道 &x 是借用对 x 的引用，而 *r 是引用 r 所指向的值即可。

继续看代码：

```
println!("The greatest common divisor of {:?} is {}",
         numbers, d);
```

迭代完 numbers 的元素，就该在标准输出流中打印结果了。println! 宏接收一个模板字符串，将后面的参数格式化之后再去替换模板字符串中对应的 {...} 形式的部分，然后把替换后的结果写入标准输出流。

C 和 C++ 要求 main 函数在程序成功结束时返回零，发生错误时返回非零退出状态码。Rust 则认为只要 main 函数返回了，程序就成功结束了。只有明确调用 expect 或 std::process::exit 才会导致程序以某个错误状态码终止。

cargo run 命令允许向程序传递参数，因此可以这样来测试命令行程序：

```
$ cargo run 42 56
   Compiling hello v0.1.0 (file:///home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.38 secs
     Running `/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
```

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `/home/jimb/rust/hello/target/debug/hello`
Usage: gcd NUMBER ...
$
```

本节使用了 Rust 标准库中的一些特性。如果想知道标准库中还包含其他什么特性，强烈建议你去看看 Rust 的在线文档。在线文档有实时搜索功能，浏览非常方便，其中甚至包含指向源代码的链接。在安装 Rust 的同时，`rustup` 命令也自动在你的计算机上安装了一份文档。可以运行如下命令在浏览器中查阅标准库的文档：

```
$ rustup doc --std
```

在线文档地址见 Rust 官方网站。

2.5　一个简单的Web服务器

可以从 crates.io 自由下载第三方依赖是 Rust 的一个优势。要在代码中使用 crates.io 上的某个包，`cargo` 命令可以搞定一切：下载正确的版本、构建代码、必要时升级。Rust 包，无论是一个库还是可执行文件，都叫 crate（意思是“木板集装箱”）。Cargo（货船）和 crates.io 都是因此而得名的。

为说明如何使用 Rust 包，本节将组装一个简单的 Web 服务器，会用到基于 `hyper` HTTP 服务器的 `iron` Web 开发框架，以及它们依赖的其他 Rust 包。如图 2-1 所示，我们的网站会提示用户输入两个数字，然后计算它们的最大公约数。

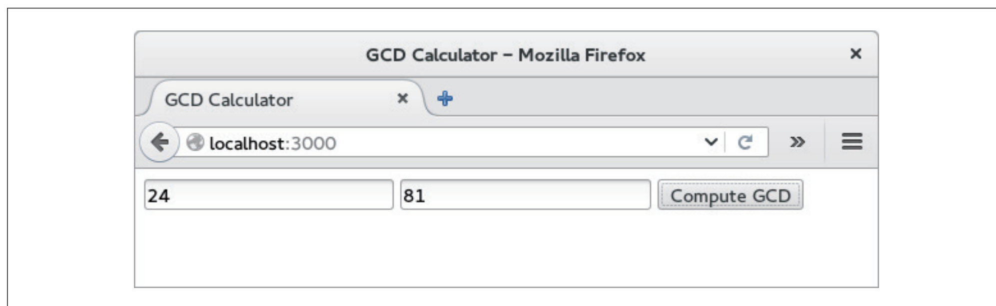


图 2-1：可以计算最大公约数的网页

首先，使用 Cargo 创建一个新包，将其命名为 `iron-gcd`：

```
$ cargo new --bin iron-gcd
     Created binary (application) `iron-gcd` project
$ cd iron-gcd
$
```

然后，修改新项目的 Cargo.toml 文件，加上要使用的包，结果如下：

```
[package]
name = "iron-gcd"
```

```
version = "0.1.0"
authors = ["You <you@example.com>"]

[dependencies]
iron = "0.5.1"
mime = "0.2.3"
router = "0.5.1"
urlencoded = "0.5.0"
```

在 Cargo.toml 的 `[dependencies]` 部分，每一行都列出了一个 crates.io 中拥有的包名，以及我们想要使用的版本。当然，crates.io 上的这几个包很可能有了比这里更新的版本，但通过在这里指定代码实测的版本，可以保证即使这些包有了新版本，代码仍然可以编译。关于 Rust 包的版本管理，第 8 章将详细讨论。

注意，这里只需直接写出要使用的包名，而这些包的依赖统统由 `cargo` 负责引入。

初版的 Web 服务器非常简单，只响应一个网页，其中包含可以输入数值的输入框。在 iron-gcd/src/main.rs 中写入以下代码：

```
extern crate iron;
#[macro_use] extern crate mime;

use iron::prelude::*;
use iron::status;

fn main() {
    println!("Serving on http://localhost:3000...");
    Iron::new(get_form).http("localhost:3000").unwrap();
}

fn get_form(_request: &mut Request) -> IronResult<Response> {
    let mut response = Response::new();

    response.set_mut(status::Ok);
    response.set_mut(mime!(Text/Html; Charset=Utf8));
    response.set_mut(r#"
        <title>GCD Calculator</title>
        <form action="/gcd" method="post">
          <input type="text" name="n"/>
          <input type="text" name="n"/>
          <button type="submit">Compute GCD</button>
        </form>
    "#);

    Ok(response)
}
```

代码开头是两个 `extern crate` 指令，分别将 Cargo.toml 文件中指定的 `iron` 和 `mime` 引入程序中。`extern crate mime` 这一行前头的 `#[macro_use]` 属性会提醒 Rust，我们打算使用这个包导出的宏。

之后，我们使用 `use` 声明导入了这两个包的一些公有特性，其中，`use iron::prelude::*` 会把 `iron::prelude` 模块中所有的公有名称直接暴露在代码中。一般来说，明确指定要

使用的特性名称更可取，比如像 iron::status 这样。不过按照约定，如果模块的名字叫 prelude，那就说明它所导出的特性是使用该包的任何用户都可能要用到的一些通用特性。因此这里在 use 指令中使用通配符 * 更合理一些。

我们的 main 函数很简单，它会打印一条消息告诉我们怎么连接到服务器，调用 Iron::new 来创建一个服务器，然后再设置让服务器监听本机 TCP 的 3000 端口。这里把 get_form 函数传给了 Iron::new，表示服务器应该用这个函数来处理所有请求。稍后还会改的。

get_form 函数则接收一个可修改的引用（&mut），指向一个 Request 类型的值，表示需要处理的 HTTP 请求。虽然当前这个处理函数并没有用到它的 _request 参数，但后面的处理函数会用到。此时，在参数名字前面加一个下划线 _ 是告诉 Rust 预期不会使用它，就不要因为这个输出警告信息了。

在这个函数体内，我们构建了一个 Response 值。set_mut 方法根据参数的类型决定设置响应的哪一部分，因此每次调用 set_mut 都会设置 response 的某个部分：传入 status::Ok 设置 HTTP 状态码，传入内容的媒体类型（使用从 mime 包导入的 mime! 宏）设置 Content-Type 首部，传入字符串设置响应体。

考虑到响应文本包含很多双引号，这里使用了 Rust 的“原始字符串”语法，即在 r（raw）后跟一个或多个 # 和一个双引号，然后是字符串内容，最后以另一个双引号及相同个数的 # 结尾。原始字符串中的任何字符都无须转义，包括双引号。事实上，原始字符串中出现的以 \ 开头的转义序列（如 \"）也不会被认为是转义序列。为避免歧义，通过在双引号两侧添加更多的 #，总是可以明确标识字符串的结束位置。

最后，函数的返回类型 IronResult<Response> 是前面遇到的 Result 类型的另一种变体，它的两个值分别可能是 Ok(r) 和 Err(e)，前者包含成功的响应值 r，后者包含某种错误值 e。在函数体末尾，我们构建了要返回的值 Ok(response)，并利用“最后一个表达式”语法隐式指定了函数的返回值。

写完了 main.rs，可以使用 cargo run 命令来完成运行程序所需的一切工作：获取必需的依赖包、编译、构建、链接，然后启动程序：

```
$ cargo run
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading iron v0.5.1
 Downloading urlencoded v0.5.0
 Downloading router v0.5.1
 Downloading hyper v0.10.8
 Downloading lazy_static v0.2.8
 Downloading bodyparser v0.5.0
...
   Compiling conduit-mime-types v0.7.3
   Compiling iron v0.5.1
   Compiling router v0.5.1
   Compiling persistent v0.3.0
   Compiling bodyparser v0.5.0
   Compiling urlencoded v0.5.0
   Compiling iron-gcd v0.1.0 (file:///home/jimb/rust/iron-gcd)
     Running `target/debug/iron-gcd`
Serving on http://localhost:3000...
```

程序启动后，在浏览器中打开指定的 URL，就可以看到图 2-1 所示的页面了。

可惜现在点击“Compute GCD”还没有用，只能重定向到 http://localhost:3000/gcd，结果还是显示相同的页面（事实上，此时访问服务器的任何 URL 结果都一样）。接下来就解决这个问题，为此要使用 `Router` 类型将不同的路径关联到不同的处理程序。

首先来安排一下，最好无须限定名就能直接使用 `Router`。那就要在 iron-gcd/src/main.rs 中添加以下声明：

```
extern crate router;
use router::Router;
```

Rust 程序员通常习惯于把所有 `extern crate` 和 `use` 声明都集中起来放到文件的顶部，实际上这不是必需的。Rust 允许这些声明以任何顺序出现，只要它们所在的嵌套层次没错就可以。（有个例外：宏定义和以 `#[macro_use]` 属性标记的 `extern crate` 必须在使用之前声明。）

接下来把 `main` 函数改成这样：

```
fn main() {
    let mut router = Router::new();

    router.get("/", get_form, "root");
    router.post("/gcd", post_gcd, "gcd");

    println!("Serving on http://localhost:3000...");
    Iron::new(router).http("localhost:3000").unwrap();
}
```

先创建了一个 `Router`，为两个特定路径配置好处理函数，然后再将这个 `Router` 传给 `Iron::new` 作为请求处理程序。返回的 Web 服务器则会根据 URL 路径决定调用哪个处理函数。

现在可以写 `post_gcd` 函数了：

```
extern crate urlencoded;

use std::str::FromStr;
use urlencoded::UrlEncodedBody;

fn post_gcd(request: &mut Request) -> IronResult<Response> {
    let mut response = Response::new();

    let form_data = match request.get_ref::<UrlEncodedBody>() {
        Err(e) => {
            response.set_mut(status::BadRequest);
            response.set_mut(format!("Error parsing form data: {:?}\n", e));
            return Ok(response);
        }
        Ok(map) => map
    };

    let unparsed_numbers = match form_data.get("n") {
        None => {
```

```
            response.set_mut(status::BadRequest);
            response.set_mut(format!("form data has no 'n' parameter\n"));
            return Ok(response);
        }
        Some(nums) => nums
    };

    let mut numbers = Vec::new();
    for unparsed in unparsed_numbers {
        match u64::from_str(&unparsed) {
            Err(_) => {
                response.set_mut(status::BadRequest);
                response.set_mut(
                    format!("Value for 'n' parameter not a number: {:?}\n",
                            unparsed));
                return Ok(response);
            }
            Ok(n) => { numbers.push(n); }
        }
    }

    let mut d = numbers[0];
    for m in &numbers[1..] {
        d = gcd(d, *m);
    }

    response.set_mut(status::Ok);
    response.set_mut(mime!(Text/Html; Charset=Utf8));
    response.set_mut(
        format!("The greatest common divisor of the numbers {:?} is <b>{}</b>\n",
                numbers, d));
    Ok(response)
}
```

这个函数中的大部分代码是 `match` 表达式，C、C++、Java 和 JavaScript 程序员可能不熟悉它，但写过 Haskell 和 OCaml 的人会觉得似曾相识。前面提到过，`Result` 的值要么是包含成功值 `s` 的 `Ok(s)`，要么是包含错误值 `e` 的 `Err(e)`。对某个 `Result` 值 `res` 而言，可以使用像下面这样的 `match` 表达式来检查它是哪种结果，并访问其中的值：

```
match res {
    Ok(success) => { ... },
    Err(error)  => { ... }
}
```

这是一个类似 C 中 `if` 或 `switch` 语句的条件表达式：如果 `res` 是 `Ok(v)`，则执行第一个分支，其中变量 `success` 设置为 `v`；如果 `res` 是 `Err(e)`，则执行第二个分支，变量 `error` 设置为 `e`。这里的 `success` 和 `error` 是各自分支的局部变量。整个 `match` 表达式的值则是执行的那个分支返回的值。

使用 `match` 表达式，程序必须先检查 `Result` 是哪种变量，然后才能访问它的值。因此我们永远不会错误地把一个失败的值当作成功的值。这正是 `match` 表达式精妙的地方。在 C 和 C++ 中，忘记检查错误码或空指针是一个常见的错误。而在 Rust 中，这类错误会在编译时

被捕获到。这个简单的手段大幅提升了程序的可用性（usability）。

Rust 支持以包含值的变体创建类似 `Result` 的自定义类型，然后使用 `match` 表达式来对其分解求值。Rust 称这种自定义类型为**枚举**（enum），有的语言可能会称其为**代数数据类型**（algebraic data type）。第 10 章将详细讨论枚举。

理解了 `match` 表达式，`post_gcd` 的结构也就清楚了。

- 调用 `request.get_ref::<UrlEncodedBody>()` 将请求体解析为一个查找表，这个查找表保存着查询参数名到参数值的数组的映射。如果解析失败，则通过响应向客户端报告错误。这个方法调用中的 `::<UrlEncodedBody>` 是一种**类型参数**（type parameter），表示要取得 `Request get_ref` 的哪一部分。在此，`UrlEncodedBody` 类型指的是作为 URL 编码的查询字符串解析的请求主体。下一节会详细介绍类型参数。
- 在这个查找表中，找到参数名为 `"n"` 的值，其中保存着通过 HTML 表单输入的数值。但这个值不是单个字符串，而是一个字符串向量，因为查询参数名是会重复的。
- 接着遍历这个字符串向量，将每个字符串解析为一个无符号 64 位整数。如果任何一个字符串解析失败，则返回相应的失败页。
- 最后，像之前那个例子一样计算最大公约数，并构建一个响应来描述结果。这里的 `format!` 宏使用与 `writeln!` 和 `println!` 宏一样的字符串模板，但它会返回一个字符串值，而不是把字符串写入输出流。

剩下的就是前面写的 `gcd` 函数了。有了这些代码，就可以终止之前运行的服务器，重新构建并重新启动程序了：

```
$ cargo run
   Compiling iron-gcd v0.1.0 (file:///home/jimb/rust/iron-gcd)
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/iron-gcd`
Serving on http://localhost:3000...
```

再打开 http://localhost:3000，输入一些数值，单击“Compute GCD”按钮，就真的可以看到结果了（如图 2-2 所示）。

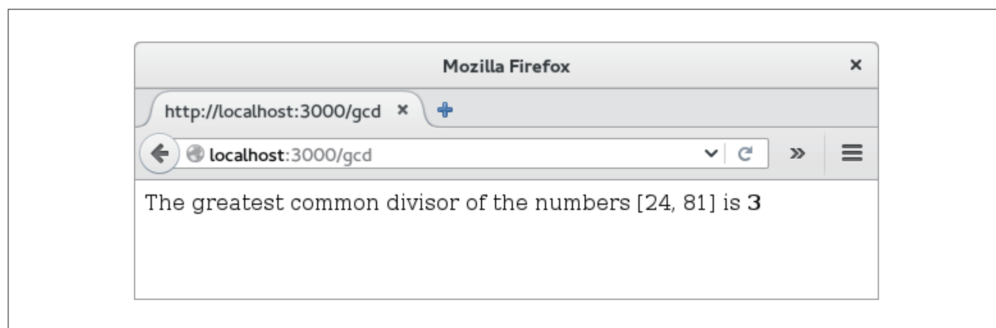


图 2-2：显示计算得到的最大公约数的 Web 页面

2.6 并发

支持并发编程是 Rust 的一大优势。确保 Rust 程序没有内存错误的规则，同样可以确保线程只能在避免数据争用的前提下共享内存。举几个例子来说明。

- 在使用互斥量协调多线程修改共享数据结构的情况下，Rust 可以保证只有持有锁的线程才能存取数据，而且操作完成后会自动释放锁。在 C 和 C++ 中，互斥量与它所保护数据的这种关系只能靠注释来描述。
- 在多个线程间共享只读数据时,Rust 可以保证程序不会意外修改该数据。在 C 和 C++ 中，类型系统也有助于起到保护作用，但很容易出错。
- 在把某个数据结构的所有权由一个线程转移到另一个线程时，Rust 可以保证前一个线程确确实实让渡了所有的权力。而在 C 和 C++ 中，只能靠程序员自己检查来保证前一个线程不会再碰该数据。如果保证不了，那后果则要取决于处理器缓存里当时有什么，或者最近对内存执行了多少次写操作。这些我们都经历过。

本节将带大家编程第二个多线程程序。

恐怕不少人还没有意识到，本书前面已经写了一个多线程程序了。在求最大公约数的那个 Web 应用里，实现服务器的 Web 框架 Iron 使用了线程池来运行处理函数。如果服务器同时收到多个请求，那么它可能会同时启动多个线程来运行 `get_form` 和 `post_gcd` 函数。听到这些，可能有人会很担心，因为我们在写这两个函数时，压根没想过并发的事。别紧张，Rust 保证这么做不会出问题，无论服务器有多复杂。只要你的程序编译通过，就不用担心数据争用了。所有 Rust 函数都是线程安全的。

本节要写的多线程程序用来绘制曼德布洛特（Mandelbrot）集合，也就是在复平面上构成分形的点的集合。绘制曼德布洛特集合属于所谓的“尴尬并行”（embarrassingly parallel）算法，因为线程间的通信模式简单到让人尴尬。第 19 章将讨论更复杂的模式。不过，这个例子虽然简单，却也能演示一些根本的问题。

首先，创建一个新的 Rust 项目：

```
$ cargo new --bin mandelbrot
     Created binary (application) `mandelbrot` project
```

所有代码都将写到 mandelbrot/src/main.rs 里，当然也需要在 mandelbrot/Cargo.toml 中添加一些依赖。

在讨论并发曼德布洛特实现之前，需要先搞清楚要做什么计算。

2.6.1 到底什么是曼德布洛特集合

看代码的时候，最好能联想到代码具体在做什么。因此，我们稍微偏离一点主题，复习一些纯数学的知识：先从最简单的情形开始，然后逐步增加复杂度，直到最终揭示出曼德布洛特集合的核心算法。

下面是一个无穷循环，使用了 Rust 专门为此编写的语法——`loop` 语句：

```
fn square_loop(mut x: f64) {
    loop {
        x = x * x;
    }
}
```

实际运行中，Rust 发现任何地方都没有用到 x，因此不会计算它的值。但就眼下而言，假设前面的代码会像写的那样运行。那么想一下，x 的值会如何变化？对任何小于 1 的数求平方会得到更小的数，即结果趋近于零。1 的平方还是 1。对任何大于 1 的数求平方会得到更大的数，即结果趋近于无穷。负数的平方会变成正数，此后就像前面几种情形一样了（如图 2-3 所示）。

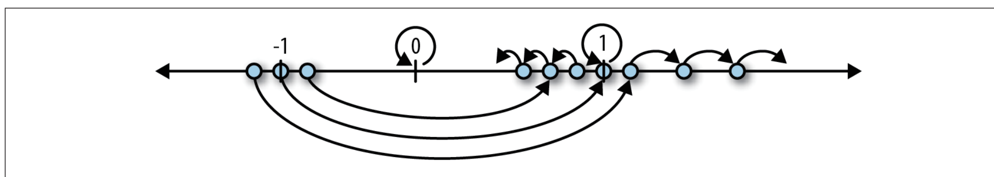


图 2-3：重复计算一个数的平方

因此，根据传给 square_loop 的值，x 要么会趋近于零，要么一直是 1，要么会趋近于无穷。

现在再看一个不太一样的循环：

```
fn square_add_loop(c: f64) {
    let mut x = 0.;
    loop {
        x = x * x + c;
    }
}
```

这一次，x 从 0 开始，每次迭代都会在求平方之后加上一个 c。如此一来，就不容易看出 x 的变化趋势了。别急，实验表明，如果 c 大于 0.25 或小于 –2.0，则 x 最终会趋近于无穷大；否则，x 就会待在邻近 0 的某个地方。

继续增加一点复杂度：在循环中把 f64 值换成复数。crates.io 上的 num 包提供了可以使用的复数类型，因此把它加到 Cargo.toml 文件的 [dependencies] 部分。目前为止，这个文件的全部内容如下（稍后还要添加其他依赖）：

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["You <you@example.com>"]

[dependencies]
num = "0.1.27"
```

现在来编写循环的倒数第二个版本：

```
extern crate num;
use num::Complex;
```

```
#[allow(dead_code)]
fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0 };
    loop {
        z = z * z + c;
    }
}
```

惯例是用 `z` 表示复数，所以这里重命名了循环变量。表达式 `Complex { re: 0.0, im: 0.0 }` 是基于 `num` 包的 `Complex` 类型表示复数 0 的写法。`Complex` 是一个 Rust 结构体类型，定义如下：

```
struct Complex<T> {
    /// 复数的实数（real）部分
    re: T,

    /// 复数的虚数（imaginary）部分
    im: T
}
```

以上代码定义了一个名为 `Complex` 的结构体，这个结构体有两个字段：`re` 和 `im`。而且，`Complex` 还是一个**泛型**（generic）结构体，代码中跟在类型名后面的 `<T>` 可以理解为“任何类型 `T`”。比如，`Complex<f64>` 就表示其 `re` 和 `im` 字段均为 `f64` 值的复数，`Complex<f32>` 则使用 32 位浮点值。基于这个定义，表达式 `Complex { re: R, im: I }` 会生成一个 `re` 字段初始化为 `R`、`im` 字段初始化为 `I` 的 `Complex` 值。

`num` 包负责处理对 `Complex` 值的 `*`、`+` 以及其他算术操作，因此这个函数的其他代码跟之前一样，只不过现在计算的不再是实数轴上的点，而是复平面上的点了。第 12 章将介绍如何让 Rust 操作符处理自定义类型的数据。

好了，简短的纯数学回顾之旅至此就结束了。曼德布洛特集合就是这样一组复数 `c` 的集合：对于任意 `c` 值，`z` 都不会接近无穷大。我们最初的简单平方循环很容易断言：任何大于 1 或小于 –1 的数都会很快接近无穷大。每次循环多了一个 `+ c` 也加大了预测难度：前面说过，大于 0.25 或小于 –2.0 的 `c` 值会导致 `z` 接近无穷大。把这个游戏扩展到复数领域，则会得到奇异又迷人的图案，那正是我们想要绘制出来的。

复数 `c` 有实部 `c.re` 和虚部 `c.im`，我们将其看作平面直角坐标系中的 `x` 和 `y` 坐标，且如果 `c` 属于曼德布洛特集合，就将该点绘制为黑色；否则就绘制为其他更浅的颜色。那么对图像中的每个像素，都必须基于复平面上对应的点来运行前面的循环，根据它会飞向无穷远还是永远围绕圆心旋转来决定这个像素的颜色。

无穷循环要花费时间，但可以用两个技巧来应对。首先，放弃运行起来没完的思路，而是只运行有限次，仍然可以得到近似的集合。至于运行多少次，取决于我们希望绘制的边界的精度。其次，事实表明，如果 `z` 离开了以原点为圆心、半径为 2 的圆，那么它最终一定会飞向离原点无穷远。

下面就是最终版的循环，也是整个程序的核心：

```
extern crate num;
use num::Complex;

/// 确定c是否属于曼德布洛特集合，最多循环limit次
///
/// 如果c不是成员，就返回Some(i)，其中i是在z离开以原点为圆心、
/// 半径为2的圆时循环的次数。如果c是成员（更准确地说，若达到
/// 循环上限尚未证明c不是成员），则返回None
fn escape_time(c: Complex<f64>, limit: u32) -> Option<u32> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        z = z * z + c;
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
    }

    None
}
```

这个函数有两个参数：`c` 是一个复数，我们要测试它是否属于曼德布洛特集合；`limit` 是循环次数的上限，到这个次数就认为 `c` 应该是成员。

函数的返回值是一个 `Option<u32>`。Rust 标准库是这样定义 `Option` 类型的：

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option` 是一个**可枚举类型**，简称**枚举**，因为其定义枚举了该类型值的几种可能变体：对任意类型 `T`，`Option<T>` 类型的值要么是 `Some(v)`（其中 `v` 是 `T` 类型的值），要么是 `None`（表示没有 `T` 类型的值可用）。跟前面讨论过的 `Complex` 类型一样，`Option` 也是一个泛型类型，即可以使用 `Option<T>` 来表达任何类型 `T` 的一个可选值。

在这里，`escape_time` 返回一个 `Option<u32>`，用以表示 `c` 是否在曼德布洛特集合中，如果不在，则返回知道这个结果循环了多少次。如果 `c` 不在集合中，`escape_time` 会返回 `Some(i)`，其中 `i` 就是 `z` 离开半径为 2 的圆时循环的次数。否则，`c` 显然在集合中，`escape_time` 返回 `None`。

```
for i in 0..limit {
```

前面的例子展示了迭代命令行参数和向量元素的 `for` 循环，而这个 `for` 循环迭代的是一个整数范围：从 0 到 `limit`（不包含）。

`z.norm_sqr()` 方法调用返回 `z` 到原点距离的平方。为确定 `z` 是否离开了半径为 2 的圆，不计算平方根，而用距离的平方与 4.0 比较会更快。

恐怕你已经注意到了，我们用 `///` 在前面的函数定义中标记了注释行，而 `Complex` 结构体成员上方的注释同样以 `///` 开头。这种注释叫作**文档注释**，`rustdoc` 实用工具知道如何解析文档注释以及它们描述的代码，并生成在线文档。Rust 标准库的文档就是以这种方式写成的。第 8 章将详细介绍文档注释。

程序的其余部分还需要确定集合的哪一部分以什么分辨率绘制，并将工作分配给多个线程以加快计算速度。

2.6.2 解析成对的命令行参数

程序需要几个命令行参数来控制要生成图像的分辨率，以及图像要展示的曼德布洛特集合的范围。由于命令行参数都具有约定的格式，因此可以写一个函数来解析它们：

```
use std::str::FromStr;

/// 解析字符串s，格式为一对坐标值，如"400x600"或"1.0,0.5"
///
/// 特别地，s的格式应该是"<左值><分隔符><右值>"的形式，其中<分隔符>
/// 就是separator参数传入的字符，而<左值>和<右值>都是字符串，可以通过
/// T::from_str来解析
///
/// 如果s的格式没错，就返回Some<(x, y)>。如果解析出错，则返回None
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None
            }
        }
    }
}

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("",        ','), None);
    assert_eq!(parse_pair::<i32>("10,",     ','), None);
    assert_eq!(parse_pair::<i32>(",10",     ','), None);
    assert_eq!(parse_pair::<i32>("10,20",   ','), Some((10, 20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ','), None);
    assert_eq!(parse_pair::<f64>("0.5x",    'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5, 1.5)));
}
```

这个 `parse_pair` 是一个**泛型函数**（generic function）：

```
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
```

函数名后面的 `<T: FromStr>` 表示“任何实现了 `FromStr` 特型的类型 `T`”。这个泛型声明让我们一次性定义了一整套函数：`parse_pair::<i32>` 是解析成对的 `i32` 值的函数，`parse_pair::<f64>` 是解析成对的浮点值的函数，等等。这跟 C++ 中的函数模板非常类似。Rust 程序员称 `T` 为 `parse_pair` 的**类型参数**（type parameter）。在使用泛型函数时，Rust 通常可以帮我们推断类型参数，因而没必要像前面测试代码中那样把类型都写出来。

函数的返回类型是 `Option<(T, T)>`，即要么是 `None`，要么是值 `Some((v1, v2))`，其中，`(v1, v2)` 是包含两个 `T` 类型值的元组。由于没有显式返回语句，因此 `parse_pair` 函数的返

回值就是函数体中最后（也是唯一）一个表达式的值：

```
match s.find(separator) {
    None => None,
    Some(index) => {
        ...
    }
}
```

String 类型的 find 方法会从字符串中搜索与 separator 匹配的字符。如果 find 返回 None，则意味着字符串里没有分隔符字符，整个 match 表达式即求值为 None，表示解析失败。否则，我们将 index 作为字符串中分隔符的位置。

```
match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
    (Ok(l), Ok(r)) => Some((l, r)),
    _ => None
}
```

从这里可以看到 match 表达式的威力。此处要匹配的参数是一个元组表达式：

```
(T::from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

表达式 &s[..index] 和 &s[index + 1..] 是分别位于分隔符前后的字符串片段。与类型参数 T 关联的 from_str 函数接收这两个字符串片段，并尝试将它们解析为 T 类型的值，最后得到结果的二元组。下面是要匹配的模式：

```
(Ok(l), Ok(r)) => Some((l, r)),
```

这个模式只匹配元组的两个元素都是 Result 的 Ok 变体的情况，表示二者均解析成功。如果模式匹配，Some((l, r)) 就是 match 表达式的值，也就是函数的返回值。

```
_ => None
```

通配符模式 _ 匹配任何东西，但忽略其值。如果到了这个分支，那么说明 parse_pair 解析失败了，因此求值结果为 None，同样也作为这个函数的返回值。

有了 parse_pair，那编写一个解析浮点坐标值对的函数就简单了，这个函数返回 Complex<f64> 值：

```
/// 将由逗号分隔的一对浮点数值解析为一个复数
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None
    }
}

#[test]
fn test_parse_complex() {
    assert_eq!(parse_complex("1.25,-0.0625"),
               Some(Complex { re: 1.25, im: -0.0625 }));
    assert_eq!(parse_complex(",-0.0625"), None);
}
```

这个 `parse_complex` 函数调用了 `parse_pair`。如果坐标解析成功，就构建一个 `Complex` 值；如果失败，则消息也会传给调用者。

如果仔细看，你会发现这里在构建 `Complex` 值时使用了简写方式。因为使用同名变量初始化结构体字段的情况很常见，所以 Rust 支持以 `Complex { re, im }` 的简单形式代替 `Complex { re: re, im: im }`。这一点模仿了 JavaScript 和 Haskell 中的类似语法。

2.6.3 像素到复数的映射

我们的程序需要对接两个相关的坐标空间，即要把输出图像的每个像素都对应到复平面上的一个点。这两个空间的对应关系取决于要绘制的是曼德布洛特集合中的哪一部分，以及图像的分辨率（由命令行参数决定）。下面这个函数负责从**图像空间**到**复数空间**的转换：

```
/// 给定输出图像中一个像素的行和列，对应到复平面上的一个点
///
/// bounds是一个元组，值为以像素计量的图像的宽和高
/// pixel是(列,行)元组，表示图像中一个特定的像素
/// upper_left和lower_right参数是复平面中的两个点，
/// 指定了图像涵盖的区域
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: Complex<f64>,
                  lower_right: Complex<f64>)
    -> Complex<f64>
{
    let (width, height) = (lower_right.re - upper_left.re,
                           upper_left.im - lower_right.im);
    Complex {
        re: upper_left.re + pixel.0 as f64 * width  / bounds.0 as f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as f64
        // 这里为什么要用减法？pixel.1越往下越大，而虚部越往上越大
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 100), (25, 75),
                              Complex { re: -1.0, im:  1.0 },
                              Complex { re:  1.0, im: -1.0 }),
               Complex { re: -0.5, im: -0.5 });
}
```

图 2-4 展示了 `pixel_to_point` 执行的计算。

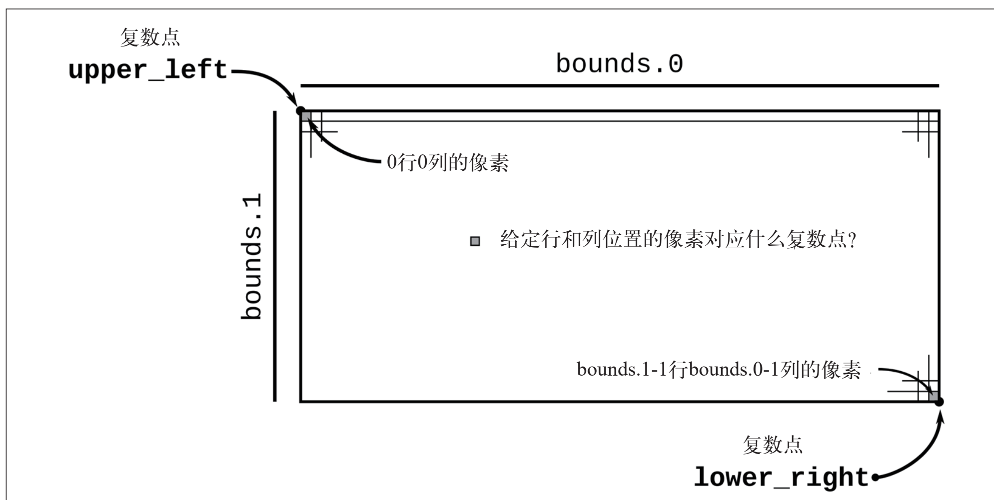


图 2-4：复平面与图像像素的对应关系

pixel_to_point 的代码只涉及简单计算，因此这里就不详细介绍了。不过，有几个地方还是要说一下。这种形式的表达式表示引用元组的元素：

```
pixel.0
```

这表示引用元组 pixel 的第一个元素。

```
pixel.0 as f64
```

这是 Rust 的类型转换语法：把 pixel.0 转换为 f64 值。与 C 和 C++ 不同，Rust 通常会拒绝数值类型的隐式转换，要转换就必须写出来。虽然略显啰唆，但明确指出怎么转换、何时转换还是非常有用的。隐式的整数转换虽然看起来没问题，但历史经验表明，它一直是很多 C 和 C++ 代码中问题和安全漏洞之源。

2.6.4　绘制集合

要绘制曼德布洛特集合，对于图像中的每个像素，只需给复平面中对应的点应用 escape_time，并根据结果确定像素的颜色即可：

```
/// 将矩形区域内的曼德布洛特集合渲染为像素保存在缓冲区
///
/// bounds参数给出缓冲区pixels的宽度和高度，后者的每个字节都保存一个
/// 灰阶像素。upper_left和lower_right参数指定与像素缓冲区中左上角和
/// 右下角的点对应的复平面上的点
fn render(pixels: &mut [u8],
          bounds: (usize, usize),
          upper_left: Complex<f64>,
          lower_right: Complex<f64>)
{
    assert!(pixels.len() == bounds.0 * bounds.1);
```

```
    for row in 0 .. bounds.1 {
        for column in 0 .. bounds.0 {
            let point = pixel_to_point(bounds, (column, row),
                                       upper_left, lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}
```

此时此刻，这些应该没有什么看不懂的。

```
pixels[row * bounds.0 + column] =
    match escape_time(point, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
```

如果 `escape_time` 说 `point` 属于曼德布洛特集合，就将对应像素渲染为黑色（`0`）。否则，脱离圆形越晚的点颜色就越深。

2.6.5 写出图像文件

`image` 包不仅提供了读写各种格式图片的功能，还提供了一些基本的图像操作功能。特别是它内置了一个 PNG 图像文件格式的编码器，我们的程序要用它保存最终计算的结果。要使用 `image`，在 Cargo.toml 文件的 `[dependencies]` 部分加上下面这行代码：

```
image = "0.13.0"
```

声明这个依赖后，就可以继续写：

```
extern crate image;

use image::ColorType;
use image::png::PNGEncoder;
use std::fs::File;

/// 把缓冲区中的pixels（大小由bounds指定）写到
/// 名为filename的文件中
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{
    let output = File::create(filename)?;

    let encoder = PNGEncoder::new(output);
    encoder.encode(&pixels,
                   bounds.0 as u32, bounds.1 as u32,
                   ColorType::Gray(8))?;
    Ok(())
}
```

这个函数的操作非常直观：打开一个文件，把图像写进去。我们传给编译器的实际的像素数据 pixels，它的宽和高来自 bounds，最后一个参数表示如何解释 pixels 中的字节。ColorType::Gray(8) 的意思是每个字节表示一个 8 位灰度值。

这些都没什么。这个函数有意思的地方是错误处理。如果遇到错误，则需要把错误回传给调用者。如前所述，Rust 容易出错的函数应该返回 Result 值，要么是表示成功的 Ok(s)，要么是表示失败的 Err(e)，其中 s 和 e 分别是成功值和错误码。那 write_image 的成功和错误是什么类型？

如果一切顺利，write_image 函数不会返回有用的值，有用的数据都写到文件里了。因此其成功类型就是**基元**（unit）类型 ()，因为这种类型只有一个值，也是 ()。基元类型有点像 C 和 C++ 中的 void。

如果发生错误，可能是 File::create 不能创建文件引起的，也可能是 encoder.encode 无法将图像写入文件造成的，I/O 操作就会返回一个错误码。File::create 返回的是 Result<std::fs::File, std::io::Error>，而 encoder.encode 返回的是 Result<(), std::io::Erro>，即二者的错误类型一样，都是 std::io::Error。为此，write_image 只要使用相同的错误类型就好了。

考虑一下对 File::create 的调用。如果成功打开的 File 值 f 返回 Ok(f)，那么 write_image 就会进一步向 f 中写入图像数据。但如果 File::create 返回包含错误码 e 的 Err(e)，那么 write_image 应该立即以 Err(e) 作为它自己的返回值。同时，对 encoder.encode 的调用也必须同样处理：失败则立即返回，传出错误码。

使用 ? 操作可以简化这些检查。原来要像这样全部写出来：

```
let output = match File::create(filename) {
    Ok(f) => { f }
    Err(e) => { return Err(e); }
};
```

现在，等价却更简洁的写法如下：

```
let output = File::create(filename)?;
```



试图在 main 函数中使用 ? 是初学者常犯的错误。因为 main 函数没有返回值，所以这种写法无效。正确的做法是使用 Result 的 expect 方法。? 操作符只对那些返回 Result 的函数有用。

其实这里还有一种简写语法可以使用。因为针对某种类型 T 返回 Result<T, std::io:: Error> 很常见（常见于涉及 I/O 操作的函数），所以 Rust 标准库为它定义了一个简写语法。在 std::io 模块中，有如下定义：

```
// std::io::Error类型
struct Error { ... };

// std::io::Result类型，等价于通常的Result，
// 但专门以std::io::Error作为错误类型
type Result<T> = std::result::Result<T, Error>
```

如果使用 std::io::Result 声明把以上定义引入作用域，那么 write_image 的返回类型就可以写成更简短的 Result<>。这种简写形式在你阅读 std::io、std::fs 和其他地方中函数的文档时经常可以看到。

2.6.6　并发的曼德布洛特程序

最后，所有逻辑已经各就各位，接下来要考虑 main 函数了，此时可以利用并发提升性能。首先来看一看非并发版本，比较简单：

```
use std::io::Write;

fn main() {
    let args: Vec<String> = std::env::args().collect();

    if args.len() != 5 {
        writeln!(std::io::stderr(),
                 "Usage: mandelbrot FILE PIXELS UPPERLEFT LOWERRIGHT")
            .unwrap();
        writeln!(std::io::stderr(),
                 "Example: {} mandel.png 1000x750 -1.20,0.35 -1,0.20",
                 args[0])
            .unwrap();
        std::process::exit(1);
    }
    let bounds = parse_pair(&args[2], 'x')
        .expect("error parsing image dimensions");
    let upper_left = parse_complex(&args[3])
        .expect("error parsing upper left corner point");
    let lower_right = parse_complex(&args[4])
        .expect("error parsing lower right corner point");

    let mut pixels = vec![0; bounds.0 * bounds.1];

    render(&mut pixels, bounds, upper_left, lower_right);

    write_image(&args[1], &pixels, bounds)
        .expect("error writing PNG file");
}
```

收集命令行参数，保存到一个 String 向量中。然后逐个解析它们并开始计算 。

```
let mut pixels = vec![0; bounds.0 * bounds.1];
```

宏调用 vec![v; n] 会创建一个 n 元素长的向量，每个元素的初始值都是 v。因此前面的代码会创建一个元素都为 0 的向量，长度为 bounds.0 * bounds.1，其中 bounds 是解析命令行参数得到的图像分辨率。如图 2-5 所示，我们会把这个向量作为一个矩形数组，保存 1 字节大小的灰度像素值。

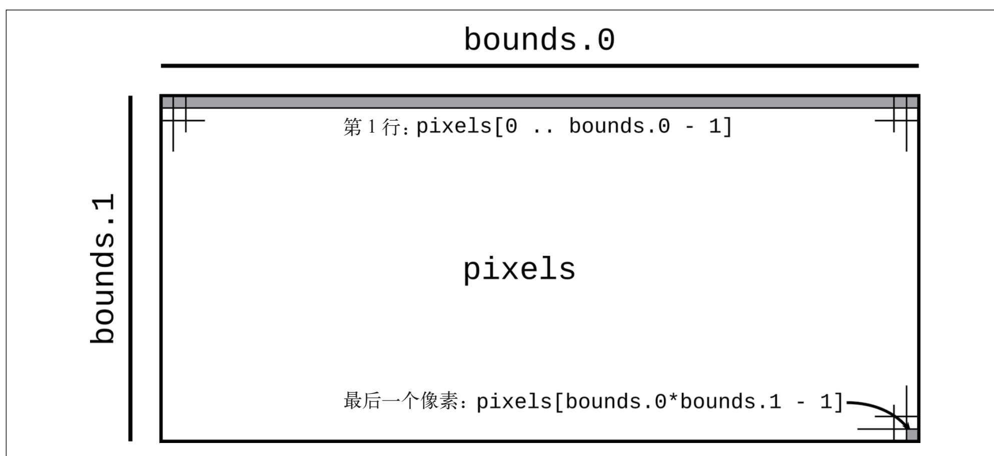


图 2-5：使用向量作为矩形的像素数组

接下来的关键代码是：

```
render(&mut pixels, bounds, upper_left, lower_right);
```

这是调用 render 函数来实际计算图像。表达式 &mut pixels 借用了像素缓冲区的一个可修改的引用，允许 render 用计算得到的灰度值填充它，这一切都是在 pixels 还是这个向量所有者的前提下进行的。接下来传入的参数决定图像的尺寸和我们选择绘制的复平面矩形。

```
write_image(&args[1], &pixels, bounds)
    .expect("error writing PNG file");
```

最后，把像素缓冲区的数据以 PNG 文件的形式写到磁盘上。这里传入的是对缓冲区的一个共享（不可修改）引用，因为 write_image 不需要修改缓冲区的内容。

要把整个计算分散到多个处理器，最自然的方式是把图像分成几块，一个处理器负责一块，并让每个处理器为分配给它的像素着色。为简单起见，我们把图像切分成水平的长条，如图 2-6 所示。在所有处理器完成计算之后，就可以把像素写入磁盘了。

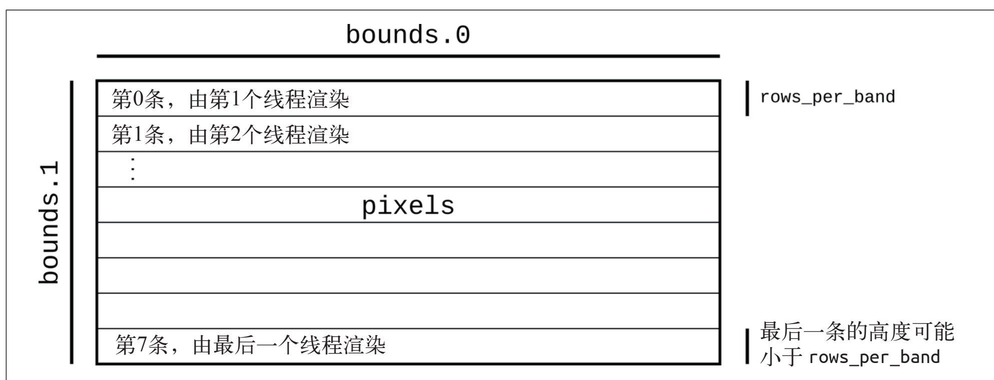


图 2-6：把像素缓冲区分成长条，以便并行渲染

crossbeam 包提供了很多实用的并发辅助功能，包括这里正好需要的**受限线程**（scoped thread）。要使用它，必须在 Cargo.toml 文件中添加下面这行代码：

```
crossbeam = "0.2.8"
```

然后，必须在 main.rs 文件顶部加上下面这行代码：

```
extern crate crossbeam;
```

接下来找到调用 render 的那行代码，把它替换成以下代码：

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;

{
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
    crossbeam::scope(|spawner| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = rows_per_band * i;
            let height = band.len() / bounds.0;
            let band_bounds = (bounds.0, height);
            let band_upper_left =
                pixel_to_point(bounds, (0, top), upper_left, lower_right);
            let band_lower_right =
                pixel_to_point(bounds, (bounds.0, top + height),
                               upper_left, lower_right);

            spawner.spawn(move || {
                render(band, band_bounds, band_upper_left, band_lower_right);
            });
        }
    });
}
```

像以前一样，一步一步来分析：

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;
```

这里声明了要使用 8 个线程[2]。之后又计算了每一条应该包含多少像素行。因为每一条的高度是 rows_per_band，而整个图像的总宽度是 bounds.0，所以长条区域的像素数量就是 rows_per_band * bounds.0。为了保证所有条加起来可以覆盖整个图像，我们给行数加 1，以免受高度不能被线程数（threads）整除的影响。

```
let bands: Vec<&mut [u8]> =
    pixels.chunks_mut(rows_per_band * bounds.0).collect();
```

这里把像素缓冲区分成了长条。缓冲的 chunks_mut 方法返回一个迭代器，可以产生可修改、不重叠的缓冲区切片，其中每个切片包含 rows_per_band * bounds.0 个像素。换句话说，也就是有 rows_per_band 个完整的像素行。chunks_mut 产生的最后一个切片包含的

注 2：num_cpus 包提供了一个函数，返回当前系统上可用的 CPU 数量。

行数较其他切片可能会少一些，但每一行包含的像素数是相同的。最后，这个迭代器的 `collect` 方法会构建一个向量，用于保存这些可修改、不重叠的切片。

现在可以使用 `crossbeam` 库了：

```
crossbeam::scope(|spawner| { ... });
```

参数 `|spawner| { ... }` 是一个 Rust **闭包**（closure）表达式。闭包就是一个可以被当作函数调用的值。在此，`|spawner|` 是参数列表，而 `{ ... }` 就是函数体。注意，跟使用 `fn` 声明的函数不同，不需要声明闭包的参数类型，Rust 会推断闭包的参数以及返回值的类型。

在这里，`crossbeam::scope` 调用了这个闭包，并传给 `spawner` 参数一个值，以便闭包可以通过它来创建新线程。`crossbeam::scope` 函数会等待所有线程都执行完成后再返回自身。这样，Rust 就可以确定这些线程在超出作用域之后，不会再访问它们对应的 `pixels` 的部分，同时也让我们知道当 `crossbeam::scope` 返回，就意味着图像的计算完成了。

```
for (i, band) in bands.into_iter().enumerate() {
```

这行代码是在迭代像素缓冲区的长条。`into_iter()` 迭代器会给予每次迭代的循环体对一个切片的专一所有权，确保每次只有一个线程可以对其执行写操作。具体细节将在第 5 章介绍。然后，`enumerate` 适配器会产生元组，包含每一个向量元素及其索引。

```
let top = rows_per_band * i;
let height = band.len() / bounds.0;
let band_bounds = (bounds.0, height);
let band_upper_left =
    pixel_to_point(bounds, (0, top), upper_left, lower_right);
let band_lower_right =
    pixel_to_point(bounds, (bounds.0, top + height),
                   upper_left, lower_right);
```

基于索引和每个长条的实际大小（别忘了最后一个可能比其他的短），可以按照 `render` 函数的要求，计算出一个定界盒子，这个盒子只对应缓冲区中的当前长条，而非整个图像。类似地，我们再利用 `pixel_to_point` 函数找到对应复平面的左上角和右下角。

```
spawner.spawn(move || {
    render(band, band_bounds, band_upper_left, band_lower_right);
});
```

最后，创建一个线程，运行闭包 `move || { ... }`。这个语法看起来有点怪，其实这是专门给函数体为 `{ ... }` 但没有参数的闭包设计的。前头的 `move` 关键字表示这个闭包会取得它所使用变量的所有权。特别注意，只有这个闭包可能会用到可修改的切片 `band`。

如前所述，`crossbeam::scope` 调用会确保所有线程在它返回前完成，也就是说只要它返回了，就可以把图像写入文件了。

2.6.7 运行曼德布洛特绘图器

我们的程序使用了几个外部包：`num` 用于复数运算，`image` 用于写 PNG 文件，`crossbeam` 用于创建受限的线程。以下就是包含所有依赖的最终的 Cargo.toml 文件：

```
[package]
name = "mandelbrot"
version = "0.1.0"
authors = ["You <you@example.com>"]

[dependencies]
crossbeam = "0.2.8"
image = "0.13.0"
num = "0.1.27"
```

配置好这些依赖就可以运行程序了：

```
$ cargo build --release
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling bitflags v0.3.3
   ...
   Compiling png v0.4.3
   Compiling image v0.13.0
   Compiling mandelbrot v0.1.0 (file:///home/jimb/rust/mandelbrot)
    Finished release [optimized] target(s) in 42.64 secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m1.750s
user    0m6.205s
sys     0m0.026s
$
```

这里使用 Unix 的 `time` 命令输出了程序的运行时间。注意，尽管计算图像花了 6 秒多的处理器时间，但实际运行时间小于两秒。可以验证一下，实际运行时间大部分花在了写图像上面，只要把相关代码注释掉就可以了。在我们测试这个程序的笔记本计算机上，这个并发的版本将曼德布洛特计算时间几乎减少为原来的四分之一。第 19 章将展示如何从根本上改进这个程序。

运行程序后应该会创建一个名为 mandel.png 的文件，你可以用自己的图片查看程序或 Web 浏览器打开它。如果一切顺利，应该可以看到类似图 2-7 所示的结果。

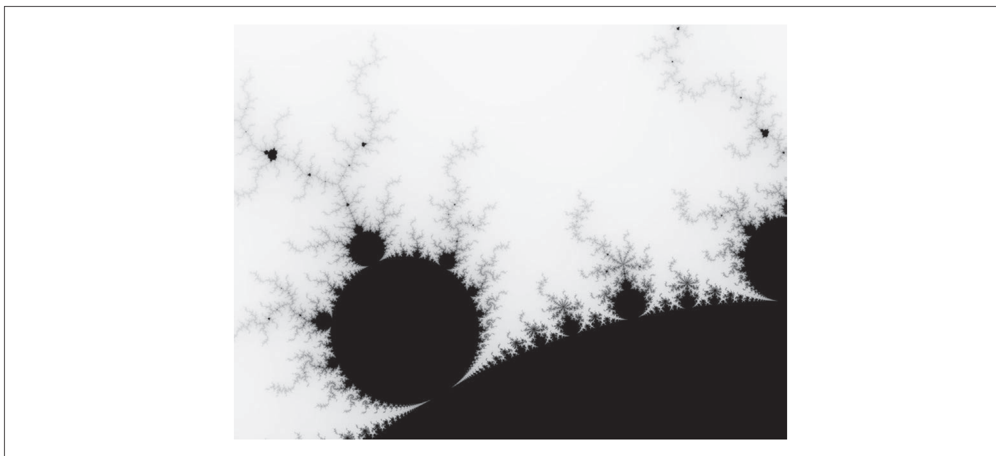


图 2-7：并行曼德布洛特程序运行的结果

2.6.8 安全无形

最后，这个并行程序可能跟用其他语言编写的没有本质区别。它们都是把像素缓冲区分成片，交给多个处理器去计算，每个处理器各自完成自己的计算，等到所有处理器都完成了，再展示结果。那么 Rust 对并发编程的支持有什么特别之处？

要知道，这里没有展示的是用 Rust 根本**写不出来**的程序。我们在本章看到的代码做到了把缓冲区正确分配给线程，但也有很多其他可能的写法做不到（因而会导致数据争用）。但 Rust 编译器的静态检查不会放过其中任何一种写法。C 或 C++ 编译器会兴致勃勃地帮你在无垠的程序空间里大海捞针般地寻找细微的数据争用逻辑，但 Rust 会提前告诉你哪里可能会出问题。

第 4 章和第 5 章将介绍 Rust 的内存安全规则。第 19 章会解释这些规则如何确保并发可靠。但为了理解这些，还需要先掌握关于 Rust 基本类型的知识，而这正是下一章的内容。

第3章

基本类型

> 世界上的书有很多很多种类型，这是合理的，因为世界上的人也有很多很多种类型，每个人都想看到不同的东西。
>
> ——Lemony Snicket（美国作家及编剧 Daniel Handler 的笔名）

Rust 的类型服务于 3 个目的。

- **安全**

  通过检查程序的类型，Rust 编译器可以防止各种常见错误。通过将空指针和未经检查的联合（union）替换成类型安全的特性，Rust 甚至可以消除很多错误，而这些错误在其他语言中常常是导致崩溃的根源。

- **效率**

  程序员对 Rust 程序如何表示内存中的值可以进行非常细粒度的控制，可以选择他们知道处理器将会高效处理的类型。程序无须因自己用不到的通用性或灵活性而折中。

- **简洁**

  程序员只需在代码中写出类型，Rust 就可以全权负责而无须过多提示。Rust 程序在类型方面通常比 C++ 程序更清晰。

Rust 在设计时没有考虑解析器或即时（JIT，Just In Time）编译器，而是选择了事先编译。换句话说，Rust 程序在执行之前会完全被转换为机器码。Rust 的类型有助于事前编译器为程序要操作的值选择更好的机器级表示，这些表示形式的性能是可预期的，因而程序员可以完全利用机器的能力。

Rust 是**静态类型的**语言，即无须实际运行程序，编译器就可以检查所有可能的执行路径，确保程序以与类型一致的方式使用每一个值。为此 Rust 可以提前捕捉到很多编程错误，而

这也是 Rust 安全保障的关键。

与 JavaScript 或 Python 等动态类型语言相比，Rust 要求写代码之前多做一些规划：要写出函数参数及返回值的类型、结构体成员的类型，以及其他一些结构体的类型。然而，实际上并没有你想象得那么麻烦，因为 Rust 提供了两个特性。

- Rust 可以根据你写出的类型**推断**其余大部分值的类型。实践中，对于某个变量或表达式，通常只有一种类型适用。此时，Rust 允许我们忽略这种类型。比如，可以像下面这样写出函数中所有值的类型：

  ```
  fn build_vector() -> Vec<i16> {
      let mut v: Vec<i16> = Vec::<i16>::new();
      v.push(10i16);
      v.push(20i16);
      v
  }
  ```

  但这样显得很乱，也有很多重复。根据函数的返回值类型，显然 `v` 的类型一定是 `Vec<i16>`，即一个包含 16 位有符号整数的向量，其他任何类型都不对。那么接着，这个向量的每个元素一定是 `i16`。这实际上也是 Rust 类型的工作原理，因此你可以这样写：

  ```
  fn build_vector() -> Vec<i16> {
      let mut v = Vec::new();
      v.push(10);
      v.push(20);
      v
  }
  ```

  这两个函数定义完全等价，无论怎么写，Rust 生成的机器码都是一样的。类型推断带来了动态类型语言才有的可读性，同时又能保证在编译时捕获错误。

- 函数可以是**泛型的**。如果函数的目的和实现足够通用，那可以将它定义为符合必要标准的任何类型集。这样一个定义就能涵盖无限种使用场景。

  在 Python 或 JavaScript 中，所有函数天生可用于任意类型。只要给定的值拥有函数运行所需的属性和方法，那它就能使用这个函数。（这个特点通常被称为**鸭子类型**：如果叫起来像鸭子，走路也像鸭子，那它就是鸭子。）然而正是这种灵活性才导致这些语言不能提前检查出类型错误，要捕获这种错误只能靠测试。

  Rust 的泛型函数为这门语言增添了一定程度的灵活性，同时还能保证在编译时捕获所有类型错误。

  不仅灵活，泛型函数还跟对应的非泛型函数一样高效。第 11 章将详细讨论泛型函数。

本章剩下的内容将自下而上地介绍 Rust 的类型，从整数和浮点值等简单的机器类型开始，然后再介绍如何将它们合成为更复杂的结构。必要时也会讨论 Rust 如何在内存中表示这些值，以及它们的性能特点。

表 3-1 总结了 Rust 中的所有类型，既有 Rust 的基本类型，也有标准库中极为常用的类型，还有一些用户定义类型的例子。

表3-1：Rust的类型

<table>
<tr><th>类　　型</th><th>说　　明</th><th>值</th></tr>
<tr><td>i8、i16、i32、i64、u8、u16、u32、u64</td><td>给定位宽的有符号和无符号整数</td><td>42、-5i8、0x400u16、0o100i16、20_922_789_888_000u64、b'*'（u8 字节字面量）</td></tr>
<tr><td>isize、usize</td><td>与机器字（32 位或 64 位）同样大的有符号和无符号整数</td><td>137、-0b0101_0010isize、0xffff_fc00usizez</td></tr>
<tr><td>f32、f64</td><td>单精度、双精度 IEEE 浮点数值</td><td>1.61803、3.14f32、6.0221e23f64</td></tr>
<tr><td>bool</td><td>布尔值</td><td>true、false</td></tr>
<tr><td>char</td><td>Unicode 字符，32 位宽</td><td>'*'、'\n'、'字'、'\x7f'、'\u{CA0}'</td></tr>
<tr><td>(char, u8, i32)</td><td>元组，允许混合类型</td><td>('%', 0x7f, -1)</td></tr>
<tr><td>()</td><td>“基元”（空）元组</td><td>()</td></tr>
<tr><td>struct S { x: f32, y: f32 }</td><td>命名字段结构体</td><td>S { x: 120.0, y: 209.0}</td></tr>
<tr><td>struct T(i32, char);</td><td>类元组结构体</td><td>T {120, 'X'}</td></tr>
<tr><td>struct E;</td><td>类基元结构体，无字段</td><td>E</td></tr>
<tr><td>enum Attend { OnTime, Late(u32)}</td><td>枚举，或代数数据类型</td><td>Attend::Late(5)、Attend::OnTime</td></tr>
<tr><td>Box<Attend></td><td>Box：拥有指向堆中值的指针</td><td>Box::new(Late(15))</td></tr>
<tr><td>&i32、&mut i32</td><td>共享和可修改的引用：非所有型指针，生命期不能超过引用的值</td><td>&s.y、&mut v</td></tr>
<tr><td>String</td><td>UTF-8 字符串，动态分配大小</td><td>"ラーメン: ramen".to_string()</td></tr>
<tr><td>&str</td><td>对 str 的引用：对 UTF-8 文本的非所有型指针</td><td>"そば: soba"、&s[0..12]</td></tr>
<tr><td>[f64; 4]、[u8; 256]</td><td>数组，固定长度，元素类型都相同</td><td>[1.0, 0.0, 0.0, 1.0]、[b' '; 256]</td></tr>
<tr><td>Vec<f64></td><td>向量，可变长度，元素类型都相同</td><td>vec![0.367, 2.718, 7.389]</td></tr>
<tr><td>&[u8]、&mut [u8]</td><td>对切片，即数组或向量某一部分的引用，包含指针和长度</td><td>&v[10..20]、&mut a[..]</td></tr>
<tr><td>&Any、&mut Read</td><td>特型对象，对任何实现了一组给定方法的值的引用</td><td>value as &Any、&mut file as &mut Read</td></tr>
<tr><td>fn(&str, usize) -> isize</td><td>函数指针</td><td>i32::saturating_add</td></tr>
<tr><td>（闭包类型没有书面形式）</td><td>闭包</td><td>|a, b| a * a + b * b</td></tr>
</table>

表 3-1 中的大多数类型会在本章介绍，其他类型将分章介绍：

- struct 类型将在第 9 章单独介绍；
- 枚举类型将在第 10 章单独介绍；
- 特型对象会放在第 11 章介绍；
- 本章介绍 String 和 &str 的基本内容，更多细节会在第 17 章介绍；
- 函数和闭包类型会在第 14 章介绍。

3.1　机器类型

Rust 类型的基础是一组固定宽度的数值类型（与几乎所有现代处理器直接在硬件中实现的类型对应），以及布尔类型和字符类型。

Rust 数值类型的命名遵循一种规律，即要同时写出位宽及其表现形式，如表 3-2 所示。

表3-2：数值类型

大小（位）	无符号整数	有符号整数	浮点数值
8	`u8`	`i8`	
16	`u16`	`i16`	
32	`u32`	`i32`	`f32`
64	`u64`	`i64`	`f64`
机器字	`usize`	`isize`	

这里，**机器字**指的是运行代码的机器上内存地址的宽度，通常是 32 位或 64 位。

3.1.1　整数类型

Rust 的无符号整数类型使用所有位表示正值和零，如表 3-3 所示。

表3-3：无符号整数

类　型	范　围
`u8`	0 到 2^8–1（0 到 255）
`u16`	0 到 2^{16}–1（0 到 65 535）
`u32`	0 到 2^{32}–1（0 到 4 294 967 295）
`u64`	0 到 2^{64}–1（0 到 18 446 744 073 709 551 615 或 1844 京）
`usize`	0 到 2^{32}–1 或 2^{64}–1

Rust 的有符号整数使用 2 的补数的表示形式，使用与无符号类型相同的位模式表示相应范围的正数和负数，如表 3-4 所示。

表3-4：有符号整数

类　型	范　围
`i8`	-2^7 到 2^7–1（–128 到 127）
`i16`	-2^{15} 到 2^{15}–1（–32 768 到 32 767）
`i32`	-2^{31} 到 2^{31}–1（–2 147 483 648 到 2 147 483 647）
`i64`	-2^{63} 到 2^{63}–1（–9 223 372 036 854 775 808 到 9 223 372 036 854 775 807）
`isize`	-2^{31} 到 2^{31}–1 或 -2^{63} 到 2^{63}–1

Rust 通常使用 `u8` 类型表示字节值。比如，从文件或套接口读取数据拿到的是 `u8` 值数据流。

与 C 和 C++ 不同，Rust 会将字符和数值类型区别对待，即 `char` 既不是 `u8` 也不是 `i8`。3.1.4 节将介绍 Rust 的 `char` 类型。

可以将 usize 和 isize 看作 C 和 C++ 中的 size_t 和 ptrdiff_t。usize 无符号，isize 有符号。它们的精度取决于目标机器的寻址空间大小：在 32 位机器上是 32 位长，在 64 位机器上是 64 位长。Rust 要求数组索引必须是 usize 值。另外，表示数组或向量的大小，或者某些数据结构中元素数量的值通常也是 usize 类型的。

在调试构建中，Rust 会检查算术操作中是否有整数溢出：

```
let big_val = std::i32::MAX;
let x = big_val + 1; // 诧异：算术操作溢出
```

在发布构建中，这个加法操作会翻转（wrap）为负值（与 C++ 不同，C++ 中的有符号整数溢出是未定义行为）。但除非你永远不想使用调试构建，否则不要指望这个行为。如果真想翻转计算结果，那么可以使用对应的方法：

```
let x = big_val.wrapping_add(1); // 没问题
```

Rust 中的整数字面量可以通过一个后缀表示类型，比如 42u8 是一个 u8 值，而 1729isize 是一个 isize。如果整数字面量中不包含类型后缀，Rust 则会根据上下文推断其类型。推断通常会找到一种唯一类型，但有时候也存在多种类型均可的情况。此时，如果可能的类型中有 i32，那么 Rust 将默认其为 i32。否则，Rust 会因存在歧义而报错。

另外，整数字面量可以使用前缀 0x、0o 和 0b 分别表示十六进制、八进制和二进制。

为了让长数值更容易认读，可以在数字间插入下划线。比如，可以把最大的 u32 值写成 4_294_967_295。在哪里插入下划线都无所谓，因此对于十六进制或二进制数值，也可以每 4 位插一个下划线，比如 0xffff_ffff。或者，也可以用下划线将数字与类型后缀分开，比如 127_u8。

表 3-5 列举了几个整数字面量的示例。

表3-5：整数字面量

字　面　量	类　　型	十进制值
116i8	i8	116
0xcafeu32	u32	51 966
0b0010_1010	推断	42
0o106	推断	70

尽管数值类型和 char 类型不一样，但 Rust 还是提供了**字节字面量**（byte literal），即用类字符字面量表示的 u8 值：b'X' 表示 ASCII 编码的字符 X，它是一个 u8 值。比如，由于字符 A 的 ASCII 编码是 65，因此 b'A' 等于 65u8。字节字面量中只能出现 ASCII 编码的字符。

有几个 ASCII 编码的字符在以字节字面量表示时需要特殊处理，不能像对待其他字符一样简单地将其放在单引号后面，否则可能导致歧义或者难以分辨。表 3-6 列出了这几个字符，在通过字节字面量表示时，需要在它们前面加一个反斜杠。

表3-6：需要转义的字符

字　　符	字符字面量	对等的数值
单引号（'）	`b'\''`	`39u8`
反斜杠（\）	`b'\\'`	`92u8`
换行	`b'\n'`	`10u8`
回车	`b'\r'`	`13u8`
制表符	`b'\t'`	`9u8`

对于难写或难读的字符，可以用十六进制写出它们的编码。在 `b'\xHH'` 形式的字节字面量中，`HH` 是两位十六进制数字，表示其数值为 `HH` 的字节。比如，“转义”控制字符的 ASCII 编码是 27，即十六进制的 `1B`，因此可以用 `b'\x1b'` 这个字节字面量表示它。因为字节字面量只是 `u8` 值的另一种写法，所以在使用它以前有必要想一想：直接使用简单的数值字面量是不是更好？像刚才那个例子，使用 `b'\x1b'` 而不是更简单的 27，唯一的理由应该是你想强调这个值表示一个 ASCII 编码。

可以使用 `as` 操作符实现整数类型之间的转换。类型转换的细节会在 6.13 节介绍，现在先来看几个例子吧：

```
assert_eq!(   10_i8  as u16,    10_u16); // 在范围内
assert_eq!( 2525_u16 as i16,  2525_i16); // 在范围内

assert_eq!(   -1_i16 as i32,    -1_i32); // 以符号填充
assert_eq!(65535_u16 as i32, 65535_i32); // 以零填充

// 超出目的类型范围的转换会得到原始值对2的N次方取模后的值，
// 其中N是目的类型的位宽。这种情况也被称为“截短”
assert_eq!( 1000_i16 as  u8,   232_u8);
assert_eq!(65535_u32 as i16,    -1_i16);

assert_eq!(   -1_i8 as u8,     255_u8);
assert_eq!(  255_u8 as i8,      -1_i8);
```

与其他值一样，整数也可以有方法。标准库提供了一些可以通过在线文档了解的基本操作。注意，文档中既包含类型自身的页面（比如，搜索“`i32` (primitive type)”），也包含该类型对应模块的页面（搜索“`std::i32`”）。比如：

```
assert_eq!(2u16.pow(4), 16);            // 乘方
assert_eq!((-4i32).abs(), 4);           // 绝对值
assert_eq!(0b101101u8.count_ones(), 4); // 数量统计
```

这几个例子中的类型后缀是必需的，因为如果没有类型，Rust 就没法找到这个值的方法。不过在实际的代码中，通常会有上下文帮助确定类型，因而就不需要这些后缀了。

3.1.2　浮点类型

Rust 支持 IEEE 单、双精度浮点类型。遵循 IEEE 754-2008 标准，这些类型包含正、负无穷，区分正、负零，还有一个**非数值**（not-a-number）值，如表 3-7 所示。

表3-7：浮点类型

类　型	精　度	范　围
f32	IEEE 单精度（至少 6 位小数）	约 -3.4×10^{38} 到 3.4×10^{38}
f64	IEEE 双精度（至少 15 位小数）	约 -1.8×10^{308} 到 1.8×10^{308}

Rust 的 `f32` 和 `f64` 对应支持 IEEE 浮点值实现的 C 和 C++ 中的 `float` 和 `double` 类型，以及 Java 中的 `float` 和 `double` 类型，Java 始终支持 IEEE 浮点值。

图 3-1 展示了浮点字面量的通用形式。

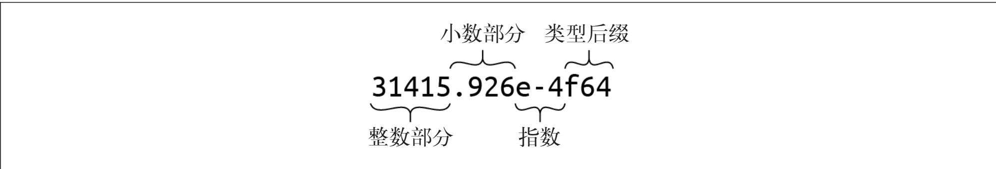


图 3-1：浮点字面量

浮点数值中除了整数部分，其他部分都是可选的，但小数部分、指数和类型后缀这三者中至少要有一个存在，这样才能将它跟整数字面量区分开。小数部分也可以只有一个小数点，因此 `5.` 也是一个有效的浮点常量。

如果浮点字面量中没有类型后缀，那么 Rust 会根据上下文推断它是 `f32` 还是 `f64`，如果两种都有可能，则默认为 `f64`。（类似地，C、C++ 和 Java 都将不带后缀的浮点字面量当作 `double` 值处理。）为了进行类型推断，Rust 将整数字面量和浮点字面量看成两种不同的类型。换句话说，浮点类型不会被推断为整数类型，反之亦然。

表 3-8 给出了一些浮点字面量的示例。

表3-8：浮点字面量

字　面　量	类　型	数　学　值
`-1.5625`	推断	$-(1\frac{9}{16})$
`2.`	推断	2
`0.25`	推断	1/4
`1e4`	推断	10 000
`40f32`	`f32`	40
`9.109_383_56e-31f64`	`f64`	约 $9.109\,383\,56 \times 10^{-31}$

标准库的 `std::f32` 和 `std::f64` 模块为 IEEE 要求的特殊值定义了常量，比如 `INFINITY`、`NEG_INFINITY`（负无穷）、`NAN`（Not A Number，非数值），以及 `MIN` 和 `MAX`（最小和最大有限值）。模块 `std::f32::consts` 和 `std::f64::consts` 提供了各种常用的数学常量，比如 `E`、`PI`，以及 2 的平方根等。

`f32` 和 `f64` 类型也提供完整的数学计算方法，比如 `2f64.sqrt()` 用于计算 2 的双精度平方根。标准库文档在“`f32` (primitive type)”和“`f64` (primitive type)”条目下给出了这些方法的说明。看几个例子：

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // 根据IEEE，正好5.0
assert_eq!(-1.01f64.floor(), -1.0);
assert!((-1. / std::f32::INFINITY).is_sign_negative());
```

同样，在真实的代码中，通常不需要写出后缀，因为根据上下文可以推断出来。然而，如果根据上下文推断不出来，可能就会出现令人不解的错误。比如，下面的代码就无法编译：

```
println!("{}", (2.0).sqrt());
```

Rust 会报错：

```
error: no method named `sqrt` found for type `{float}` in the current scope
```

这有点不太好理解，不在浮点类型上找，难道还到其他类型上去找 `sqrt` 方法吗？解决方案是写出你想要的类型，以下两种写法都行：

```
println!("{}", (2.0_f64).sqrt());
println!("{}", f64::sqrt(2.0));
```

与 C 和 C++ 不同，Rust 几乎不进行隐式数值类型转换。如果函数接收 `f64` 参数，传入 `i32` 值就会导致错误。事实上，Rust 甚至都不会隐式地将 `i16` 值转换为 `i32` 值，即使每个 `i16` 值也是 `i32` 值。不过，这里的关键词是**隐式**。使用 `as` 操作符进行**显式**转换是没有问题的，比如 `i as f64` 或 `x as i32`。不支持隐式转换有时候会让同样的 Rust 代码比 C 和 C++ 代码显得啰唆。然而，隐式整数转换经常会导致 bug 和安全漏洞也是有案可查的。根据我们的经验，Rust 要求明确写出数值类型转换，让我们多次避免了问题。6.13 节将具体解释类型转换。

3.1.3　布尔类型

Rust 的布尔类型 `bool` 按惯例有两个值：`true` 和 `false`。`==` 和 `<` 等比较操作符都产生布尔值，比如 `2 < 5` 是 `true`。很多语言在要求布尔值的上下文中允许使用其他类型值。C 和 C++ 会隐式地将字符、整数、浮点数和指针转换为布尔值，因此这些类型的值可以直接在 `if` 或 `while` 语句中用作条件。Python 允许在布尔值上下文中使用字符串、列表、字典，甚至集合，如果这些值不为空，则将它们视为 `true`。Rust 则非常严格：`if` 和 `while` 等控制结构要求它们的条件必须是 `bool` 表达式，短路逻辑操作符 `&&` 和 `||` 同样也是如此。换句话说，`if x { ... }` 不正确，必须写成 `if x != 0 { ... }` 才行。

Rust 的 `as` 操作符可以把 `bool` 值转换为整数类型：

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

不过，`as` 不会反向转换，即不能从数值类型转换为布尔值。此时，必须明确地写出可以返回布尔值的比较表达式，比如 `x != 0`。

虽然 `bool` 只需要一位即可表示，但 Rust 在内存里会使用整整一个字节作为 `bool` 值，因此可以创建一个指向它的指针。

3.1.4 字符类型

Rust 的字符类型 char 以 32 位值的形式表示单个 Unicode 字符。

Rust 使用 char 类型表示单个字符，但对字符串或文本流使用 UTF-8 编码。因此，String 将其文本表示为一个 UTF-8 字节的序列，而不是字符的数组。

字符字面量就是用单引号引起来的字符，比如 '8' 或 '!'。可以使用任意 Unicode 字符，比如 ' 錆 '（音 sabi）是英文 rust 对应的日本汉字的 char 字面量。

类似字节字面量，在字符字面量中有几个字符也需要使用反斜杠转义，如表 3-9 所示。

表3-9：需要转义的字符

字　　符	Rust字符字面量
单引号（'）	'\''
反斜杠（\）	'\\'
换行	'\n'
回车	'\r'
制表符	'\t'

如果愿意，也可以在单引号里写出相应字符的十六进制 Unicode 码点。

- 如果字符的码点范围在 U+0000 到 U+007F 之间（也就是对应的 ASCII 字符集），可以将该字符写成 '\xHH' 形式，其中 HH 是 2 位十六进制数字。比如，字符字面量 '*' 和 '\x2A' 是相等的，因为字符 * 的码点就是 42 或者十六进制的 2A。
- 任何 Unicode 字符都可写作 '\u{HHHHHH}'，其中 HHHHHH 是 1 到 6 位十六进制数字。比如，字符字面量 '\u{CA0}' 表示 Unicode 不赞成表情（Look of Disapproval）“ಠ_ಠ”中用到的坎纳达语（Kannada，印）字符“ಠ”。当然 'ಠ' 也是相同的字面量。

char 类型保存的 Unicode 码点范围只能在 0x0000 到 0xD7FF 之间，或 0xE000 到 0x10FFFF 之间。char 不能是 Unicode 代理对中的某一半（也就是码点范围不能在 0xD800 到 0xDFFF 之间），也不能是 Unicode 编码空间之外的值（也就是不能大于 0x10FFFF）。Rust 使用类型系统及动态检查来确保 char 的值始终位于许可的范围内。

Rust 从来不会隐式地在 char 与其他类型之间进行转换。如果需要，可以使用转换操作符 as 把 char 转换为整数类型，但如果目的类型小于 32 位，字符值的高位（upper bits）则会被截短：

```
assert_eq!('*' as i32, 42);
assert_eq!('ಠ' as u16, 0xca0);
assert_eq!('ಠ' as i8, -0x60); // U+0CA0被截短为8位有符号数
```

而反方向上，u8 则是唯一可以转换为 char 的整数类型。因为 Rust 希望 as 只执行低开销、高可靠的转换，但除 u8 之外的所有整数类型都可能包含不被许可的 Unicode 码点，所以这时候的转换还需要进行运行时检查。不过，标准库函数 std::char::from_u32 可以将 u32 值转换为 Option<char> 值：如果该 u32 是不被许可的 Unicode 码点，就返回 None；否则，返回 Some(c)，其中 c 就是转换后的 char 值。

标准库也为字符提供了一些有用的方法，可以通过在在线文档中搜索“`char` (primitive type)”和“`std::char`”（模块）来查看。比如：

```
assert_eq!('*'.is_alphabetic(), false);
assert_eq!('β'.is_alphabetic(), true);
assert_eq!('8'.to_digit(10), Some(8));
assert_eq!('ಠ'.len_utf8(), 3);
assert_eq!(std::char::from_digit(2, 10), Some('2'));
```

单个字符自然没有字符串和文本流那么有意思。3.5 节将详细介绍 Rust 标准的 `String` 类型和文本处理。

3.2　元组

元组（tuple）由 2 个、3 个、4 个……各种类型的值组成。元组的写法是把逗号分隔的元素序列放在一对圆括号里。比如，`("Brazil", 1985)` 是一个元组，其第一个元素是一个静态分配的字符串，第二个元素是一个整数，它的类型是 `(&str, i32)`（或者 Rust 推断出来适合 `1985` 的任何整数类型）。拿到一个元组 `t`，可以像 `t.0`、`t.1` 这样访问它的元素。

元组不太像数组。首先，元组的每个元素都可以是一个不同的类型，而数组的元素必须都是同一类型。其次，元组只允许用常量作为索引，比如 `t.4`，不能通过 `t.i` 或 `t[i]` 取得第 `i` 个元素。

Rust 代码中的函数经常通过元组来返回多个值。比如，字符串切片的 `split_at` 方法会将一个字符串切成两半，然后返回两个字符串，其声明如下：

```
fn split_at(&self, mid: usize) -> (&str, &str);
```

这个返回类型 `(&str, &str)` 就是一个包含两个字符串切片的元组。可以使用模式匹配语法把这个方法返回值的两个元素分别赋值给不同的变量：

```
let text = "I see the eigenvalue in thine eye";
let (head, tail) = text.split_at(21);
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

以下是前面代码更好理解的版本：

```
let text = "I see the eigenvalue in thine eye";
let temp = text.split_at(21);
let head = temp.0;
let tail = temp.1;
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

可以把元组看成一种最小限度的结构体类型。比如，在第 2 章曼德布洛特程序中，我们需要给函数传递图像的宽和高，然后函数才能绘制并将图像写入磁盘。为此可以声明一个包含 `width` 和 `height` 成员的结构体，但这样有点小题大做了，所以我们只传了一个元组：

```
/// 把缓冲区中的pixels（大小由bounds指定）写到名为filename的文件中
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{ ... }
```

`bounds` 参数的类型为 `(usize, usize)`，即两个 `usize` 值的元组。没错，这里直接传一个 `width` 再传一个 `height` 也可以，最终的机器码可能也差不多。但这是一个怎么写更清晰的问题。我们认为大小是一个值而不是两个，元组恰好可以表达这种意图。

另一个常用的元组类型可能会让人难以置信，它就是零元组 `()`。这种元组过去一直被称为**基元类型**（unit type），因为它只有一个值，也写作 `()`。当不存在有意义的值而上下文又要求某种类型时，Rust 会使用这种基元类型。

例如，没有返回值的函数的返回类型就是 `()`。标准库的 `std::mem::swap` 函数没有什么值得返回的值，它只是把两个参数的值交换一下。`std::mem::swap` 的声明如下：

```
fn swap<T>(x: &mut T, y: &mut T);
```

这个 `<T>` 表示 `swap` 是**泛型**函数，即任何类型 `T` 值的引用都可以用它。但这个方法签名完全省略了 `swap` 的返回类型，其实它是对以下这个返回基元类型版本的简写：

```
fn swap<T>(x: &mut T, y: &mut T) -> ();
```

类似地，前面提到的 `write_image` 返回 `Result<(), std::io::Error>` 类型，意思是这个函数会在出错时返回 `std::io::Error`，成功时则不返回值。

如果你喜欢，也可以在元组的最后一个元素后面加个逗号，比如 `(&str, i32)` 和 `(&str, i32,)` 是相同的类型，而表达式 `("Brazil", 1985)` 和 `("Brazil", 1985,)` 也相等。不仅如此，Rust 允许在函数参数、数组、结构体、枚举等任何使用逗号的地方的末尾再加一个逗号。这在人类看来可能有点怪，但在列表末尾发生了元素的增删操作之后，差异会比较容易看出来。

出于一致性的考虑，甚至可以有包含一个单一值的元组。字面量 `("lonely hearts",)` 就是只包含一个单一字符串的元组，其类型为 `(&str,)`。这时候，末尾的逗号就是必需的了，没有它则无法区分这是个元组还是简单的括号表达式。

3.3 指针类型

Rust 有几种表示内存地址的类型。

这是 Rust 在垃圾收集方面与大多数语言不一样的地方。在 Java 中，如果 `class Tree` 包含一个字段 `Tree left;`，那么 `left` 就是一个指向在其他地方创建的另一个 `Tree` 对象的引用。Java 中的对象从来不会在物理上包含另一个对象。Rust 不同，其设计目标就是保持内存占用最小化，因此 Rust 中的值默认是嵌套的，比如 `((0, 0), (1440, 900))` 会被保存为 4 个相邻的整数。如果把它保存在一个局部变量中，那它就变成了一个 4 个整数宽的局部变量。此时不会占用堆内存。

这对提升内存使用效率非常重要，但同时也带来了一个后果：当 Rust 程序中的值需要指向其他值时，就必须显式地使用指针类型。不过不用担心，在安全 Rust 程序中使用指针会受到限制以消除未定义行为，因而指针在 Rust 中比在 C++ 中更容易正确使用。

本节将讨论 3 种指针类型：引用、Box 和不安全指针。

3.3.1 引用

一个 `&String`（读作“引用字符串”）类型的值就是一个对 `String` 值的引用，而一个 `&i32` 就是一个对 `i32` 值的引用，以此类推。

理解引用最简单的办法就是把它想象成 Rust 的基本指针类型。引用可以指向任何地方的任何值，无论栈上还是堆上。表达式 `&x` 会产生一个对 `x` 的引用，用 Rust 的话说，就是它**借用了一个对 `x` 的引用**。而拿到一个引用 `r`，表达式 `*r` 引用的则是 `r` 指向的值。这跟 C 和 C++ 中的 `&` 和 `*` 非常相似。而且就像 C 指针一样，引用在超出作用域之后不会自动释放任何资源。

但与 C 指针不同的是，Rust 引用永远不会是空值，因为在安全 Rust 代码中根本不可能产生空引用。而且 Rust 引用默认是不可修改的。

- `&T`

  不可修改引用，类似 C 中的 `const T*`。

- `&mut T`

  可修改引用，类似 C 中的 `T*`。

另一个主要的区别是 Rust 会跟踪值的所有权和生成期，因此悬空指针、重复释放和指针失效之类的错误全部会在编译时被发现。第 5 章将解释 Rust 安全引用使用规则。

3.3.2 Box

在堆上分配一个值的最简单方式就是使用 `Box::new`：

```
let t = (12, "eggs");
let b = Box::new(t); // 在堆中分配一个元组
```

`t` 的类型是 `(i32, &str)`，因此 `b` 的类型是 `Box<(i32, &str)>`。`Box::new()` 会为这个元组在堆上分配足够大的内存。而当 `b` 超出作用域之后，内存会自动释放，除非 `b` 被**转移**了（比如被返回）。

转移是 Rust 处理分配到堆上的值的根本方式，第 4 章将详细解释。

3.3.3 原始指针

Rust 也有原始指针类型 `*mut T` 和 `*const T`。原始指针实际上就是 C++ 中的那种指针。使用原始指针是不安全的，因为 Rust 不会跟踪它指向哪里。为此，原始指针可能为空，也可能指向已经被释放的内存，或者指向一个现在已经保存了不同类型值的内存。C++ 中所有经典的指针错误应有尽有，敬请好自为之。

不过，对于原始指针解引用一般只能在 `unsafe` 块中进行。`unsafe` 块是 Rust 提供的可选机制，专为安全责任自负的程序员使用高级语言特性提供。如果你的代码里没用 `unsafe` 块（或者有但写的不正确），那么本书反复强调的安全保障则始终有效。关于不安全的代码，第 21 章将详细讨论。

3.4　数组、向量和切片

Rust 有 3 种在内存中表示一系列值的类型。

- 类型 `[T; N]` 表示一个 `N` 个值的数组，每个值的类型都是 `T`。数组的大小是在编译时确定的常量，也是类型自身的一部分。换句话说，不能向数组中追加新元素，也不能缩短数组。
- 类型 `Vec<T>` 叫作 `T` **类型的向量**，是一种动态分配、可扩展的类型 `T` 的值序列。向量的元素保存在堆上，因此可以随意缩放它：向其中推入新元素、追加其他向量，或者删除元素，等等。
- 类型 `&[T]` 和 `&mut [T]` 叫作 `T` **类型的共享切片**和 `T` **类型的可修改切片**，是对其他值（比如数组或向量）中部分元素序列的引用。可以把切片想象成一个指针，包含切片中第一个元素的地址和从这个元素起可以访问元素的数量。可修改切片 `&mut [T]` 允许读写元素，但不能共享；而共享切片 `&[T]` 可以由多个读者读取，但不允许修改元素。

对于这 3 种类型的值 `v`，表达式 `v.len()` 返回 `v` 中的元素数量，而 `v[i]` 引用的是 `v` 的第 `i` 个元素。第一个元素是 `v[0]`，最后一个元素是 `v[v.len() - 1]`。Rust 会检查 `i` 始终位于有效范围内；否则，表达式会导致惊异。`v` 的长度可能是零，此时尝试用任何索引访问其元素都会导致惊异。`i` 必须是一个 `usize` 值，其他任何整数类型都不能用作索引。

3.4.1　数组

数组的值有几种写法。最简单的写法是把一系列逗号分隔的值用方括号括起来：

```
let lazy_caterer: [u32; 6] = [1, 2, 4, 7, 11, 16];
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];

assert_eq!(lazy_caterer[3], 7);
assert_eq!(taxonomy.len(), 3);
```

对于常见的给一个长数组填充值的情况，可以写作 `[V; N]`：`V` 是每个元素的值，`N` 是长度。比如，`[true; 10000]` 就是一个包含 10 000 个 `bool` 元素的数组，所有元素都是 `true`：

```
let mut sieve = [true; 10000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 10000 {
            sieve[j] = false;
            j += i;
        }
    }
}
```

```
assert!(sieve[211]);
assert!(!sieve[9876]);
```

还有一种写法，即 `[0u8; 1024]`，用于表示固定大小的缓冲区（这里是 1 KB 的缓冲区，以零字节填充）。Rust 没有语法表示未初始化的数组。（通常，Rust 会保证代码永远不访问任何未经初始化的值。）

数组的长度是其类型的一部分，在编译时就确定了。如果 `n` 是一个变量，那么通过写 `[true; n]` 不会得到一个 `n` 元素的数组。如果你在运行时需要一个数组的长度可变（很常见），那就用向量。

我们看到的用于数组的方法，比如迭代元素、搜索、排序、填充、筛选等，都是切片的方法，并非数组的方法。不过，Rust 会在查找方法时隐式地把对数组的引用转换为对切片的引用，因此可以直接在数组上调用切片的任何方法：

```
let mut chaos = [3, 5, 4, 1, 2];
chaos.sort();
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```

这里的 `sort` 方法实际上是在切片上定义的，但因为 `sort` 是通过引用取得其操作数的，所以可以直接在 `chaos` 上使用它。这个调用会隐式地产生一个将整个数组作为切片的引用 `&mut [i32]`。事实上，前面提到的 `len` 方法也是一个切片方法。3.4.4 节会更详细地讨论切片。

3.4.2 向量

向量 `Vec<T>` 是分配在堆上的类型 `T` 的可缩放序列。

创建向量的方式有好几种。最简单的方式是使用 `vec!` 宏，这种语法看起来非常像创建数组字面量：

```
let mut v = vec![2, 3, 5, 7];
assert_eq!(v.iter().fold(1, |a, b| a * b), 210);
```

当然，这是一个向量，不是数组，因此可以动态地向其中添加元素：

```
v.push(11);
v.push(13);
assert_eq!(v.iter().fold(1, |a, b| a * b), 30030);
```

还可以通过模仿数组字面量的语法将一个值重复一定次数来构建向量：

```
fn new_pixel_buffer(rows: usize, cols: usize) -> Vec<u8> {
    vec![0; rows * cols]
}
```

调用 `vec!` 宏等价于调用 `Vec::new` 创建一个新的空向量，然后再向其中推入元素，这也是一种写法：

```
let mut v = Vec::new();
v.push("step");
v.push("on");
```

```
v.push("no");
v.push("pets");
assert_eq!(v, vec!["step", "on", "no", "pets"]);
```

另一种方式是基于迭代器产生的值来构建向量：

```
let v: Vec<i32> = (0..5).collect();
assert_eq!(v, [0, 1, 2, 3, 4]);
```

在使用 `collect` 时通常要写明类型（就像这里一样），因为这个方法可以构建多种集合，不仅是向量。明确了 `v` 的类型，就明确了我们想要构建的集合类型。

与数组类似，对向量也可以使用切片方法：

```
// 回文！
let mut v = vec!["a man", "a plan", "a canal", "panama"];
v.reverse();
// 合理但令人扫兴：
assert_eq!(v, vec!["panama", "a canal", "a plan", "a man"]);
```

这里的 `reverse` 方法实际是在切片上定义的，只不过这个调用隐式地从向量借用了 `&mut [&str]`，并在这个切片引用上调用了 `reverse`。

`Vec` 是 Rust 的一个基本类型，几乎任何需要动态列表的地方都会用到它，因此还有很多其他方法可以构建新向量或扩展已有向量。相关内容第 16 章将介绍。

一个 `Vec<T>` 包含 3 个值：对分配在堆上用于保存元素的缓冲区的引用、该缓冲区可以存储的元素个数（也就是容量），以及当前实际存储的元素个数（也就是长度）。当缓冲区达到其容量上限后，再给向量添加元素会导致一系列操作：分配一个更大的缓冲区，将现有内容复制过去，基于新缓冲区更新向量的指针和容量，最后释放旧缓冲区。

如果提前知道向量中需要保存的元素个数，可以使用 `Vec::with_capacity`（而不是 `Vec::new`）先创建一个有足够大缓冲区存储所有元素的向量，然后再逐个向向量中添加元素，从而避免内存的重新分配。`vec!` 宏使用了与此类似的一个技巧，因为它知道向量最终会拥有多少元素。注意，这样只是配置了向量的初始大小，如果超出了预估，那么向量还是会像往常一样增大。

很多库函数会尽量使用 `Vec::with_capacity`，避免使用 `Vec::new`。比如，在上面 `collect` 的例子中，迭代器 `0..5` 事先知道自己会产生 5 个值，而 `collect` 函数会利用它为自己返回的向量预分配正确的容量。第 15 章将深入讨论这个过程。

就像向量的 `len` 方法返回它包含的元素个数一样，它的 `capacity` 方法返回无须重新分配即可包含的元素个数：

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);
```

```
v.push(3);
assert_eq!(v.len(), 3);
assert_eq!(v.capacity(), 4);
```

你在自己代码里看到的容量可能会跟这里有出入。Vec 和系统的堆分配程序可能会向上舍入，即使在使用 with_capacity 的情况下也是如此。

可以在向量的任何位置随意插入或删除元素。不过，由于这些操作会导致插入点之后的元素全部前移或后移，因此对于长向量而言可能会比较慢。

```
let mut v = vec![10, 20, 30, 40, 50];

// 在索引3处插入35
v.insert(3, 35);
assert_eq!(v, [10, 20, 30, 35, 40, 50]);

// 删除索引1处的元素
v.remove(1);
assert_eq!(v, [10, 30, 35, 40, 50]);
```

使用 pop 方法可以删除并返回最后一个元素。更准确地说，是从 Vec<T> 中删除一个值，返回一个 Option<T>：如果向量是空的则返回 None；如果最后一个元素是 v 则返回 Some(v)：

```
let mut v = vec!["carmen", "miranda"];
assert_eq!(v.pop(), Some("miranda"));
assert_eq!(v.pop(), Some("carmen"));
assert_eq!(v.pop(), None);
```

使用 for 循环可以迭代遍历向量：

```
// 取得命令行参数并转换为字符串向量
let languages: Vec<String> = std::env::args().skip(1).collect();
for l in languages {
    println!("{}: {}", l,
             if l.len() % 2 == 0 {
                 "functional"
             } else {
                 "imperative"
             });
}
```

运行这个程序并传入几个编程语言的名称，你会受到启发：

```
$ cargo run Lisp Scheme C C++ Fortran
   Compiling fragments v0.1.0 (file:///home/jimb/rust/book/fragments)
     Running `.../target/debug/fragments Lisp Scheme C C++ Fortran`
Lisp: functional
Scheme: functional
C: imperative
C++: imperative
Fortran: imperative
$
```

最终，我们看到了对**函数式语言**的满意定义。

尽管扮演了最基本的角色，Vec 仍然是 Rust 定义的一个普通类型，没有内置在语言中。第 21 章将介绍实现这种类型的技术。

3.4.3　逐个元素地构建向量

以每次添加一个元素的方式构建向量并没有听起来那么差劲。每当向量增长到超过其缓冲区容量时，它都会选择一个比原来大一倍的新缓冲区。假设有一个向量，其缓冲区开始只有一个元素，随着元素增加，其容量也会 1、2、4、8 这样增大，直至最终增大到 2^n。想一想 2 的乘方的原理，会发现所有之前较小的缓冲区之和等于 2^n–1，与最终的缓冲区大小非常接近。因为实际元素的个数至少是缓冲区大小的一半，所以向量中的元素自始至终都不会有两个副本。

这意味着，使用 `Vec::with_capacity` 而不是 `Vec::new` 可以从速度而非算法上获得持续改进。对于较小的向量，少调用几次堆分配器性能差异还是很明显的。

3.4.4　切片

切片（slice），写作不指定长度的 `[T]`，表示数组或向量的一个范围。由于切片可以是任意长度，因此不能直接保存到变量中，也不能作为函数参数传递。切片永远只能按引用传递。

切片的引用是一个**胖指针**（fat pointer），即一个双字宽的值，保存着指向切片第一个元素的指针和切片中元素的个数。

假设运行如下代码：

```
let v: Vec<f64> = vec![0.0,  0.707,  1.0,  0.707];
let a: [f64; 4] =     [0.0, -0.707, -1.0, -0.707];

let sv: &[f64] = &v;
let sa: &[f64] = &a;
```

运行最后两行时，Rust 会自动把引用 `&Vec<f64>` 和 `&[f64; 4]` 转换为直接指向数据的切片引用。

最终，内存看起来如图 3-2 所示。

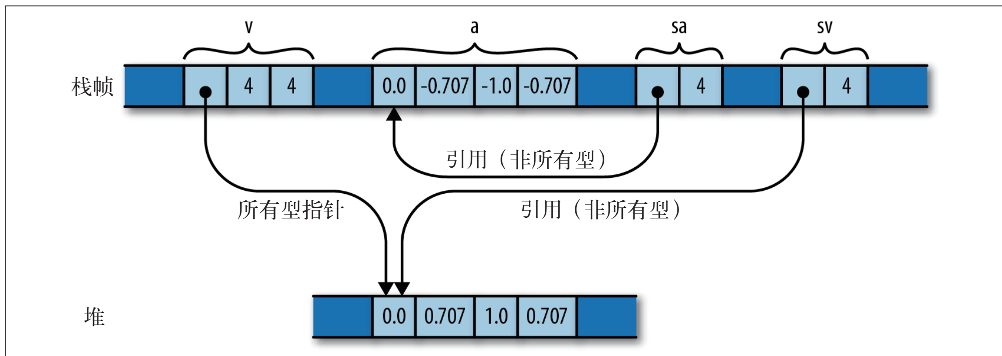


图 3-2：内存中的向量 v 和数组 a，以及引用它们的切片 sv 和 sa

普通引用是对单个值的非所有型指针，切片引用则是对多个值的非所有型指针。因此切片引用非常适合操作一串同类数据的函数，无论这串数据存储在数组、向量，抑或栈还是堆上。比如，下面这个函数打印了一个数值切片，每行打印了一个数值：

```
fn print(n: &[f64]) {
    for elt in n {
        println!("{}", elt);
    }
}

print(&v); // 操作向量
print(&a); // 操作数组
```

因为这个函数以一个切片引用作为参数，所以可以像上面代码所示的那样给它传入向量或数组的引用。事实上，很多我们以为是向量或数组的方法是定义在切片上的。比如，就地排序和反转一系列元素的 `sort` 和 `reverse` 方法实际上是切片类型 `[T]` 的方法。

在索引中使用范围可以获取对数组或向量切片的引用，或者对现有切片的切片的引用：

```
print(&v[0..2]);    // 打印v的前两个元素
print(&a[2..]);     // 从a[2]开始打印a的元素
print(&sv[1..3]);  // 打印v[1]和v[2]
```

与访问普通数组类似，Rust 也会检查此时的索引是否有效。试图借用超出数据长度范围的切片会导致惊异。

我们经常把 `&[T]` 或 `&str` 这样的引用类型称为**切片**，其实是有点偷懒了。实际上，应该称它们为**对切片的引用**。不过因为提到切片基本上都是指对它的引用，所以也就用“切片”来指代“切片引用”了。

3.5 字符串类型

熟悉 C++ 的程序员会记得该语言中有两种字符串类型。字符串字面量拥有指针类型 `const char *`。标准库也定义了一个类，即 `std::string`，用于在运行时动态创建字符串。

Rust 也采用了类似的设计。本节将展示字符串字面量的各种写法，然后再介绍 Rust 的两种字符串类型。关于字符串和文本处理的更多内容，参见第 17 章。

3.5.1 字符串字面量

字符串字面量放在一对双引号中。需要使用反斜杠转义的情形跟 `char` 字面量相同：

```
let speech = "\"Ouch!\" said the well.\n";
```

与 `char` 字面量不同的是，在字符串字面量中，单引号无须转义，双引号需要转义。

字符串可以分散到多行：

```
println!("In the room the women come and go,
    Singing of Mount Abora");
```

这个字符串字面量中的换行符和第二行开头的空白符都包含在字符串中，因此也会输出。

如果多行字符串中的一行以反斜杠结尾，那么该行的换行符和下一行开头的空白符会被删除：

```
println!("It was a bright, cold day in April, and \
    there were four of us—\
    more or less.");
```

这行代码会打印一行文本。字符串的“and”和“there”之间只有一个空格，因为代码中的反斜杠前面有一个空格，而连字符后面没有空格。

有时候，把所有反斜杠都打两遍也很烦人（经典的例子就是正则表达式和 Windows 路径）。针对这种情况，Rust 提供了**原始字符串**（raw string）语法。原始字符串以小写字母 `r` 为标记，其中的所有反斜杠和空白符都会原样包含在字符串中，转义序列无效。

```
let default_win_install_path = r"C:\Program Files\Gorillas";

let pattern = Regex::new(r"\d+(\.\d+)*");
```

要在原始字符串中包含双引号，简单地在双引号前面加个反斜杠是不行的。前面刚说过，原始字符串中的转义序列**无效**。不过也有办法，即在原始字符串的开头和末尾添加井字号标记：

```
println!(r###"
    This raw string started with 'r###"'.
    Therefore it does not end until we reach a quote mark ('"')
    followed immediately by three pound signs ('###'):
"###);
```

开头和末尾加多少个井字号都可以，只要能让 Rust 确定原始字符串在哪儿结尾就行。

3.5.2 字节字符串

字节字符串（byte string）就是带有前缀 `b` 的字符串字面量。字节字符串是 `u8` 值（即字节）的切片，不是 Unicode 文本的切片：

```
let method = b"GET";
assert_eq!(method, &[b'G', b'E', b'T']);
```

字节字符串可以使用我们刚刚看到的所有字符串语法：可以跨多行、使用转义序列、使用反斜杠连接行。原始字节字符串以 `br"` 开头。

字节字符串不能包含任意 Unicode 字符，只能是 ASCII 和 `\xHH` 转义序列。

这里 `method` 的类型是 `&[u8; 3]`，即对包含 3 个字节的数组的引用。它没有后面将要介绍的任何字符串方法。最像字符串的地方就是它的语法。

3.5.3 字符串在内存中的表示

Rust 字符串就是 Unicode 字符序列，但它在内存中不是以 `char` 数组的形式存储的。事实上，字符串在内存中使用 UTF-8 可变宽度编码存储。字符串中的每个 ASCII 字符用 1 个字

节存储，其他字符则占用多个字节。

图 3-3 展示了以下代码创建的 String 和 &str 值。

```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "ಠ_ಠ";
```

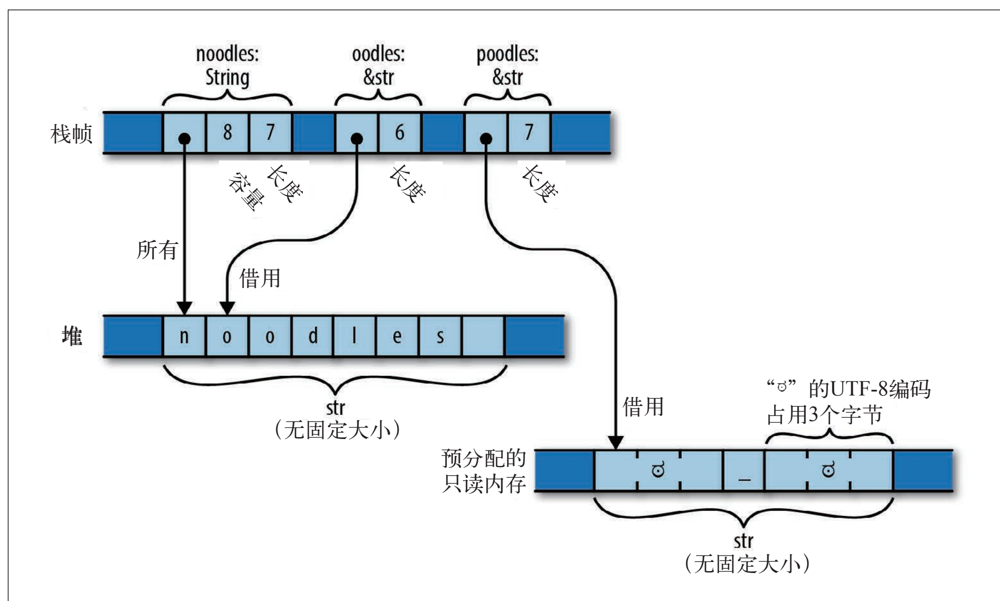


图 3-3：String、&str 和 str

String 有一个可伸缩缓冲区用于存储 UTF-8 文本。这个缓冲区分配在堆上，因此可以按需要或请求来调整大小。在这个例子中，noodels 是一个拥有 8 字节缓冲区的 String，它占用了其中 7 字节。可以把 String 想象成一个能够存储格式完好的 UTF-8 的 Vec<u8>。事实上，String 也确实是这么实现的。

&str（发音 "stir" 或叫 "字符串切片"）是对其他变量拥有的一串 UTF-8 文本的引用：它 "借用" 了这些文本。在这个例子中，oodles 是一个对 noodles 拥有的文本后 6 个字节的 &str 引用，因此它表示的是文本 "oodles"。与其他切片引用类似，&str 也是一个胖指针，包含实际数据的地址及其长度。可以把 &str 想象成就是一个 &[u8]，只不过它能存储格式完好的 UTF-8。

字符串字面量就是一个引用预分配文本的 &str，通常跟程序的机器码一起存储在只读内存中。在前面的例子中，poodles 是一个字符串字面量，指向程序执行时创建的 7 个字节，这 7 个字节在程序退出时会被释放。

String 或 &str 的 .len() 方法返回它们的长度。长度以字节而非字符度量：

```
assert_eq!("ಠ_ಠ".len(), 7);
assert_eq!("ಠ_ಠ".chars().count(), 3);
```

不要试图修改 &str，因为它是无法修改的：

```
let mut s = "hello";
s[0] = 'c';    // error: the type `str` cannot be mutably indexed
s.push('\n'); // error: no method named `push` found for type `&str`
```

如果想在运行时创建新字符串，那么要使用 String。

确实存在 &mut str 类型，但用处不大，因为几乎对 UTF-8 的任何操作都会改变其总字节长度，而切片又不可能重新分配它引用的缓冲区。事实上，&mut str 上仅有两个方法，即 make_ascii_uppercase 和 make_ascii_lowercase，根据定义它们只就地修改文本且只影响单字节字符。

3.5.4 字符串

&str 类似 &[T]，就是一个指向某些数据的胖指针。String 则类似 Vec<T>，如表 3-10 所示。

表3-10：Vec<T>与String对比

	Vec<T>	**String**
自动释放内存	是	是
可扩展	是	是
::new() 和 ::with_capacity() 静态方法	是	是
.reverse() 和 .capacity() 方法	是	是
.push() 和 .pop() 方法	是	是
范围语法 v[start..stop]	是，返回 &[T]	是，返回 &str
自动转换	&Vec<T> 到 &[T]	&String 到 &str
继承方法	继承自 &[T]	继承自 &str

类似 Vec，每个 String 也有自己分配在堆上的缓冲区，不跟任何其他 String 共享。当 String 变量超出作用域后，其缓冲区会自动释放，除非 String 的所有权已被转移。

以下几种方式可以创建 String。

- .to_string() 方法可以把 &str 转换为 String。实际上是复制字符串。

```
let error_message = "too many pets".to_string();
```

- format!() 宏跟 println!() 类似，区别在于它返回一个新 String 而不是将文本写入标准输出，并且不会自动在末尾加换行符。

```
assert_eq!(format!("{}°{:02}′{:02}″N", 24, 5, 23),
           "24°05′23″N".to_string());
```

- 字符串的数组、切片和向量有两个方法：.concat() 和 .join(sep)，可以将多个字符串拼接成一个新 String。

```
let bits = vec!["veni", "vidi", "vici"];
assert_eq!(bits.concat(), "venividivici");
assert_eq!(bits.join(", "), "veni, vidi, vici");
```

有时候，到底应该使用 &str 还是 String 总会让人困惑。第 5 章将详细讨论这个问题。目前，只要知道 &str 可以引用任何字符串的任意切片即可，无论它是（存储在可执行文件中的）字符串字面量，还是（运行时分配和释放的）String。这意味着当调用者可以使用任意一种字符串时，&str 更适合作为函数参数。

3.5.5 使用字符串

字符串支持 == 和 != 操作符。如果两个字符串包含的字符相同、顺序也相同，那它们就是相等的，无论它们是否指向内存中的同一个地址。

```
assert!("ONE".to_lowercase() == "one");
```

字符串也支持比较操作符 <、<=、> 和 >=，以及很多有用的方法和函数。通过在在线文档中搜索“str (primitive type)”或“std::str”模块可以查看（或者直接翻到第 17 章也可以）。以下是几个例子：

```
assert!("peanut".contains("nut"));
assert_eq!("ಠ_ಠ".replace("ಠ", "■"), "■_■");
assert_eq!("    clean\n".trim(), "clean");

for word in "veni, vidi, vici".split(", ") {
    assert!(word.starts_with("v"));
}
```

但要记住，Unicode 的特性决定了简单的逐个字符比较**不一定**得到想要的结果。比如，Rust 字符串 "th\u{e9}" 和 "the\u{301}" 都是 thé（法语“茶”）的有效 Unicode 表现形式。对这两个字符串，Unicode 认为无论显示和处理都应该一视同仁，但 Rust 认为它们是两个完全不同的字符串。类似地，像 < 这样的比较操作符只会简单地基于字符码点值的词典顺序进行比较。这个顺序只是有时候会跟特定用户语言和文化下的文本顺序一致。这些问题第 17 章将详细讨论。

3.5.6 其他类似字符串的类型

Rust 保证字符串是有效的 UTF-8。有时候，程序可能需要处理**不是**有效 Unicode 的字符串。这通常发生在 Rust 程序必须与其他系统互操作，而其他系统又没有采取相同强制措施的情况下。比如，在大多数操作系统中，可以轻易创建一个文件，但其文件名不是有效的 Unicode。那么当 Rust 程序碰到这种文件名时怎么办？

Rust 的解决方案是提供几个类似字符串的类型。

- 对于 Unicode 文本，使用 String 和 &str。
- 处理文件名时，使用 std::path::PathBuf 和 &Path。
- 处理根本不是字符数据的二进制数据时，使用 Vec<u8> 和 &[u8]。
- 处理以操作系统原生形式表示的环境变量名和命令行参数时，使用 OsString 和 &OsStr。
- 与使用空字符结尾字符串的 C 库互操作时，使用 std::ffi::CString 和 &CStr。

3.6　更多类型

类型是 Rust 的中枢。本书后面还会继续讨论类型，同时也会介绍一些新类型。特别地，Rust 的用户定义类型让这门语言更具特色，因为用户定义类型中会定义方法。有 3 种用户定义类型，本书会用连续 3 章介绍它们：结构体（第 9 章）、枚举（第 10 章）、特型（第 11 章）。

函数和闭包都有各自的类型，第 14 章将讨论。而标准库中定义的类型，本书各章都会有所涉及。比如，第 16 章将介绍标准的集合类型。

不过，上述这些内容都要等一等。在此之前，必须先花点时间弄清楚与 Rust 核心安全规则相关的几个概念。

第 4 章

所有权

> 我发现，Rust 强迫我掌握了之前 C/C++ 中的诸多最佳实践，而且是在编译代码之前。……我想强调的是，Rust 不是那种让你几天就可以上手，把困难 / 技术难题 / 最佳实践的事都向后推的语言。它会强迫你马上就进入严格安全的编程状态，一开始可能会有点不适应。不过，根据我的经验，这让我再次找到了我很期待的编译代码的感觉。
>
> ——Mitchell Nordine

Rust 做出了以下两个承诺，这两个承诺同时确保这门系统编程语言是安全的。

- 你来决定程序中每个值的生命期。Rust 会在你的控制下迅速释放与某个值关联的内容和其他资源。
- 即便如此，你的程序也永远不会在一个对象被释放后还使用指向它的指针。在 C 和 C++ 中，使用**悬空指针**（dangling pointer）是常见错误：运气好，程序会崩溃；运气不好，程序就有了一个安全漏洞。Rust 会在编译时捕获这些错误。

C 和 C++ 也遵守第一个承诺：你随时可以对动态分配在堆内存中的对象调用 `free` 或 `delete`。但作为交换，第二个承诺则被置之不理：保证不使用指向已释放值的指针完全是你的责任。经验表明，这个使命很难达成：在各类安全问题的数据报告中，指针误用都是一个常见问题，只要这些数据已经被收集到。

为实现第二个承诺，很多语言使用垃圾收集在所有指向某个对象的可访问指针都失效时自动释放该对象。但作为交换，你必须把控制何时释放对象的权力让渡给收集器。一般来说，垃圾收集器的脾气也不好琢磨，有时候本该释放的内存却没有释放，而原因也很难查明。如果你使用的对象表示文件、网络连接或其他系统资源，但不能确保这些资源（及其相关资源）在你认为该释放时被释放，那真的很让人头疼。

Rust 不接受上述任何一种妥协：程序员应该控制值的生命期，**同时**语言还要确保安全。可是语言设计的这个领域已经被充分探索过了，如果不在基础上做出改变，就很难带来明显的改进。

Rust 以一种非凡的方式打破僵局，即限制程序使用指针的方式。本章及下一章将具体介绍这些限制及其背后的原理。现在，只要知道你过去习惯使用的某些结构可能不适用于这些规则，因此需要另寻出路就好了。而这些限制的净效果，就是向混乱中引入足够的秩序，使 Rust 编译时检查能验证你的程序不包含影响内存安全的错误：悬空指针、重复释放、使用未初始化的内存，等等。在运行时，指针就是单纯的内存地址，跟 C 和 C++ 中的一样。区别在于你的代码已经通过了安全检查。

同样的规则也构成了 Rust 支持安全并发编程的基础。使用 Rust 精心设计的线程原语（primitive），确保你的代码正确使用内存的规则，同样也能避免数据争用。Rust 程序中的一个 bug 不会导致一个线程破坏另一个线程的数据，也不会在无关的地方引入难以复现的错误。多线程代码固有的不确定性行为通过互斥量、消息通道、原子值等这些专门设计的特性得以隔离，而不是以赤裸裸的内存引用存在。C 和 C++ 的多线程代码饱受诟病，Rust 则打了一个漂亮的翻身仗。

Rust 打赌自己能够成功的赌注，同时也是这门语言的根基所在，就是即便有了这些限制，你仍然会发现几乎对任何任务来说，这门语言都足够灵活。而为了获得这些好处（系统性消灭内存管理和并发的各类 bug）必须做出的编程风格上的改变是绝对值得的。之所以看好 Rust，恰恰是因为我们拥有多年的 C 和 C++ 背景。如果让我们选，那想都不用想，就选 Rust。

Rust 的规则你可能在其他任何语言中都没见过。依我们看，学习 Rust 的关键就在于理解和利用好这些规则。本章先在分析其他语言中同样问题的基础上给出 Rust 的规则。然后再进一步解释 Rust 的规则。最后会谈一谈某些异常和准异常。

4.1 所有权

如果你读过比较多的 C 或 C++ 代码，那可能看到过有注释解释说某个类的实例**拥有**它所指向的一些其他对象。这种情况通常意味着拥有对象来决定什么时候释放它所有的对象，即在所有者被销毁的时候，它也会销毁与之相关的资源。

比如，有如下 C++ 代码：

```
std::string s = "frayed knot";
```

这里的字符串 s 在内存中的表示如图 4-1 所示。

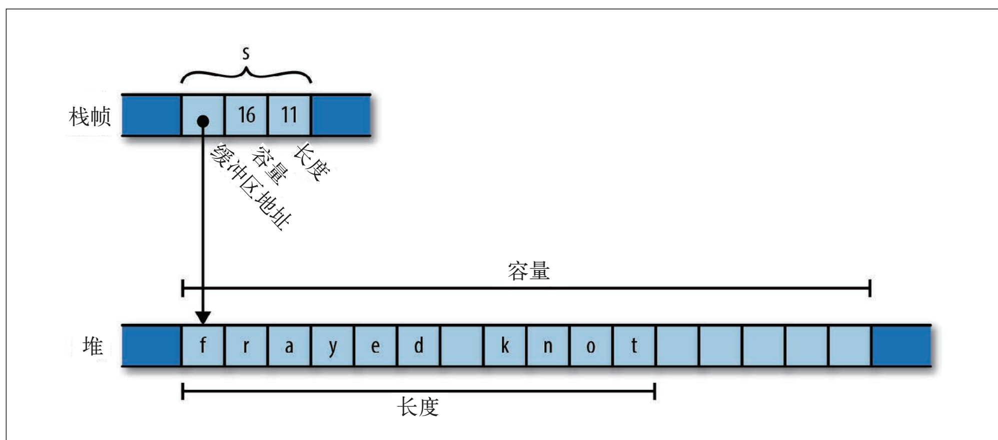


图 4-1：栈中的 C++ std::string 值指向堆中的缓冲区

在此，实际的 std::string 对象本身始终是 3 个字长，包括一个指向堆缓冲区的指针、缓冲区的总容量（也就是在分配更大的缓冲区之前，它所能容纳的文本的最大长度）和当前保存的文本的长度。它们都是 std::string 类的私有字段，这个字符串的用户是访问不到的。

std::string 拥有自己的缓冲区：当程序销毁这个字符串时，字符串的析构函数会释放该缓冲区。过去，有的 C++ 库会让多个 std::string 值共享同一个缓冲区，并通过引用计数来决定何时释放缓冲区。较新版本的 C++ 规范实际上禁止这么做，因此所有比较新的 C++ 库都使用如图所示的实现。在这种情况下，虽然其他代码也可以创建一个指向同一缓冲区的临时指针，但该代码必须负责在内存所有者销毁所有的对象之前先销毁那些指针。比如，可以创建一个指向 std::string 缓冲区的字符的指针，但在整个字符串被销毁后，这个指针就无效了，你必须确保自己不再使用这个指针。所有者决定其拥有对象的生命期，其他代码必须遵从所有者的决定。

Rust 从这些注释中提炼出规则，并明确在语言中加入相应限制。在 Rust 中，每个值都只有一个决定其生命期的所有者。当这个所有者被释放［用 Rust 的话说是**被清除**（dropped）］时，其所有的值也会被清除。这些规则旨在让开发者通过浏览代码就能看出任何值的生命期，并赋予开发者对这些值的生命期的控制权。而这都是作为一门系统编程语言应该提供的。

变量拥有它的值。当控制流离开声明变量的代码块时，变量被清除，因此变量的值也会随之被清除。比如：

```
fn print_padovan() {
    let mut padovan = vec![1,1,1];  // 分配
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
}                                   // 清除
```

这里变量 padovan 的类型是 std::vec::Vec<i32>，即 32 位整数的向量。在内存中，padovan 最终的值如图 4-2 所示。

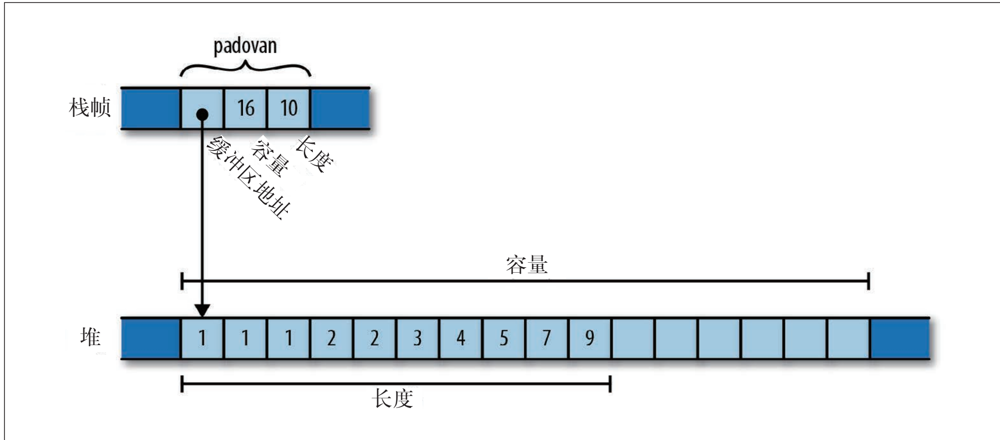


图 4-2：栈中的 Vec<32> 指向堆中的缓冲区

这跟前面 C++ std::string 值的内存表示非常类似，只不过这里缓冲区中的元素是 32 位数值，而不是字符。注意，表示 padovan 指针、容量和长度的 3 个字直接保存在 print_padovan 函数的栈帧中，只有向量的缓冲区分配在堆上。

与前面的字符串 s 类似，这里的向量拥有保存其元素的缓冲区。当变量 padovan 在函数底部超出作用域时，程序会清除这个向量。由于向量拥有其缓冲区，因此该缓冲区也一并被清除。

Rust 的 Box 类型可以作为所有权的另一个例子。Box<T> 是一个指针，其指向存储在堆中的 T 类型的值。调用 Box::new(v) 会在堆上分配相应的空间，并将值 v 转移进去，最后返回一个指向该堆空间的 Box。因为 Box 拥有它指向的空间，所以当 Box 被清除时，其指向的空间也会被清除。

比如，可以像下面这样在堆中分配一个元组：

```
{
    let point = Box::new((0.625, 0.5));  // 分配point
    let label = format!("{:?}", point);  // 分配label
    assert_eq!(label, "(0.625, 0.5)");
}                                        // 两者都被清除
```

程序在调用 Box::new 时，会在堆上为两个 f64 值的元组分配空间，然后将参数 (0.625, 0.5) 转移过去，最后返回指向该空间的指针。当控制流到达 assert_eq! 调用时，栈帧如图 4-3 所示。

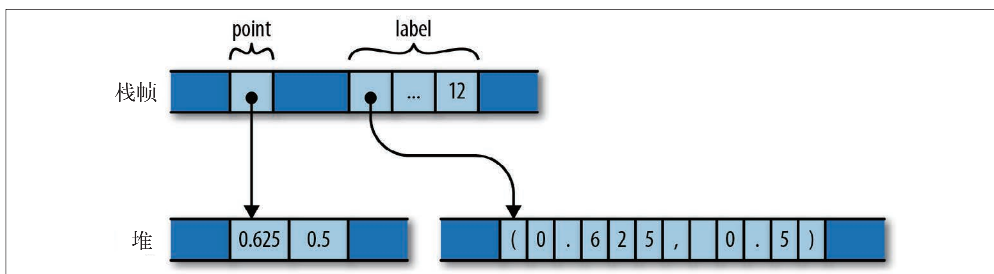


图 4-3：两个局部变量，其分别拥有各自的堆空间

这里的栈帧本身保存着变量 `point` 和 `label`，它们分别引用自己拥有的堆空间。当这两个变量被清除时，对应的堆空间也会被清除。

与变量拥有自己的值一样，结构体也拥有自己的字段。相应地，元组、数组和向量则拥有自己的元素：

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{}, born {}", composer.name, composer.birth);
}
```

这时，`composers` 是一个 `Vec<Person>`，即一个结构体的向量，每个结构体分别保存一个字符串和数值。在内存中，最终的 `composers` 值如图 4-4 所示。

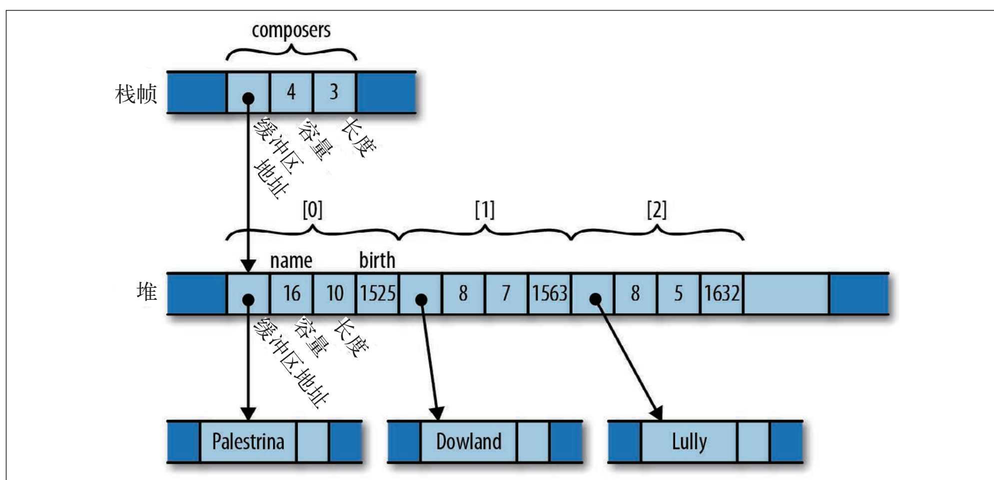


图 4-4：复杂一点的所有权树

这里涉及多个所有权关系，但每个都很清晰：composers 拥有一个向量，而向量拥有自己的元素，每个元素都是一个 Person 结构体，而每个结构体又拥有自己的字段，其中字符串字段拥有自己的文本。当控制流离开声明 composers 的作用域时，程序会清除自己的值，并清除与之相关的所有内存空间。假如这张图里有其他集合，比如 HashMap 或 BTreeSet，意思也是一样的。

此时，后退一步，想一想目前展示的所有权关系会带来什么结果。每个值都只有一个所有者，因此决定何时清除它很容易。但是，一个值可能拥有多个其他值，比如 composers 向量就拥有多个元素。而这些值同样可能再拥有自己的值，比如 composers 的每个元素都拥有一个字符串，每个字符串又各自拥有自己的文本。

换句话说，可以通过**树**来描述所有权关系：你的所有者是你的父节点，你拥有的值是你的子节点，每个树的根节点则是某一个变量。当控制流超出了这个变量所在的作用域时，整个树都会被清除。仍以 composers 的所有权树为例，但这个树并不是搜索数据结构中的“树”或者 HTML 文档的 DOM 元素树的概念，而是基于混合类型构建的一棵树，且严格遵守 Rust 的单一所有者规则，禁止任何形式的结构再连接（否则就不是纯粹的树了）。Rust 程序中的每一个值都是某棵树的成员，都可以追溯到某个根变量。

Rust 程序通常根本不会明确地清除值，即不会像在 C 和 C++ 中使用 free 和 delete 那样。在 Rust 中，清除某个值某种意义上是通过将其从所有权树中删除实现的。比如，离开了某个变量的作用域或者从向量中删除了一个元素之类的。此时，Rust 确保相应的值及其一众资源都会被正确清除。

从某种角度看，Rust 似乎不如其他语言强大：因为几乎所有其他实用的编程语言都允许开发者构建任意形式的对象图谱，对象之间可以相互任意引用。但恰恰因为 Rust 在这方面不够强大，才使得它对程序所能进行的分析可以更强大。Rust 的安全性应该正是源自这种代码中实体关系的可追踪性。这是前面提到的 Rust“激进赌注”的一部分。事实上，Rust 宣称即便施加了这样的限制，人们照样可以在解决问题时拥有足够的灵活性，找到足够完美的解决方案。

虽说如此，上述规则仍然有点太严苛了。Rust 从几个方面对此进行了扩展。

- 可以把值从一个所有者转移到另一个所有者，从而方便构建、重塑和销毁关系树。
- 标准库提供了基于引用计数的指针类型 Rc 和 Arc，使用它们可以在满足某些限制条件的前提下将值指定给多个所有者。
- 对一个值，可以“借用其引用”。引用是生命期有限的非所有指针。

以上策略都为所有权模型注入了灵活性，同时也支持了 Rust 的承诺。接下来，我们将依次介绍它们，但引用会放在下一章介绍。

4.2 转移

在 Rust 中，对多数类型而言，给变量赋值、给函数传值或从函数返回值这样的操作不会复制值，而是**转移**（move）值。所谓转移，就是原来的所有者让渡这个值的所有权给目标所有者，并变成未初始化状态。然后，由目标所有者控制这个值的生命期。Rust 程序会以每

次一个值、每次转移一个的形式构造和拆解复杂的结构。

有人可能惊讶于 Rust 会改变如此基础性操作的含义。没错，赋值在这个历史当口应该具有相当明确的含义。可是，如果你仔细分析几种不同语言处理赋值的方式，就会发现现实中已经存在明显不同的风格了。比较下来，你会发现 Rust 之所以选择如此，其用意和重要性也是显而易见的。

先来看看下面的 Python 代码：

```
s = ['udon', 'ramen', 'soba']
t = s
u = s
```

每个 Python 对象都带有一个引用计数，记录当前正在引用它的值的数量。因此，在给 s 赋值后，上面这段代码的状态如图 4-5 所示（注意有些字段被忽略了）。

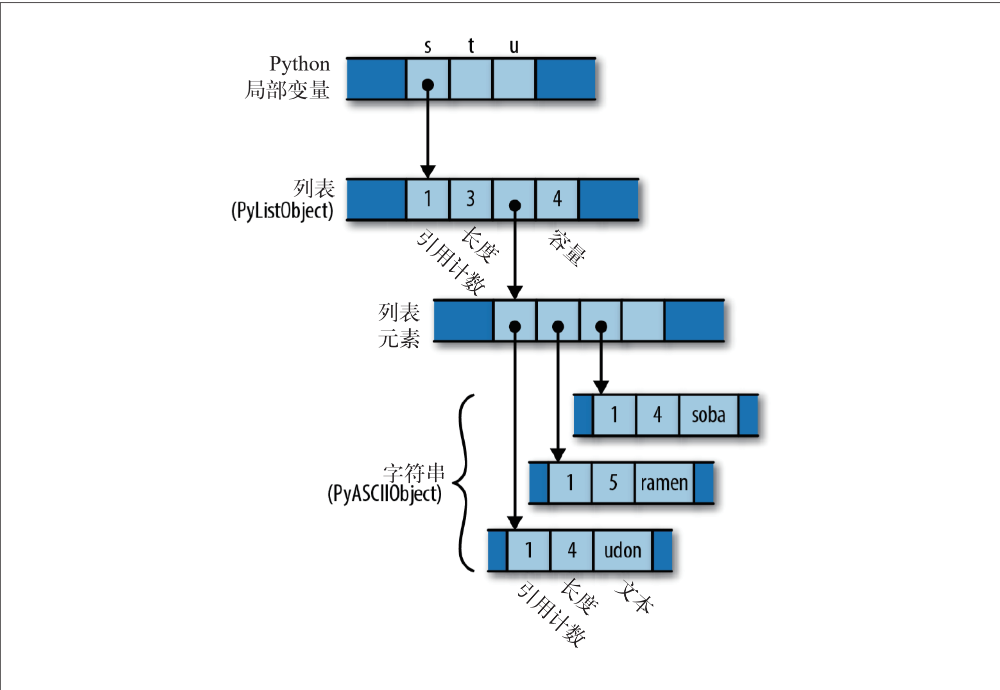


图 4-5：Python 字符串列表在内存中的表示

因为只有 s 指向列表，所以列表的引用计数为 1。而又因为列表是唯一指向字符串的对象，所以每个字符串的引用计数也是 1。

当程序执行到 t 和 u 的赋值时会怎么样？ Python 对这两个赋值的实现是让目标变量指向与源变量相同的对象，同时递增该对象的引用计数。因此这段程序最终的状态如图 4-6 所示。

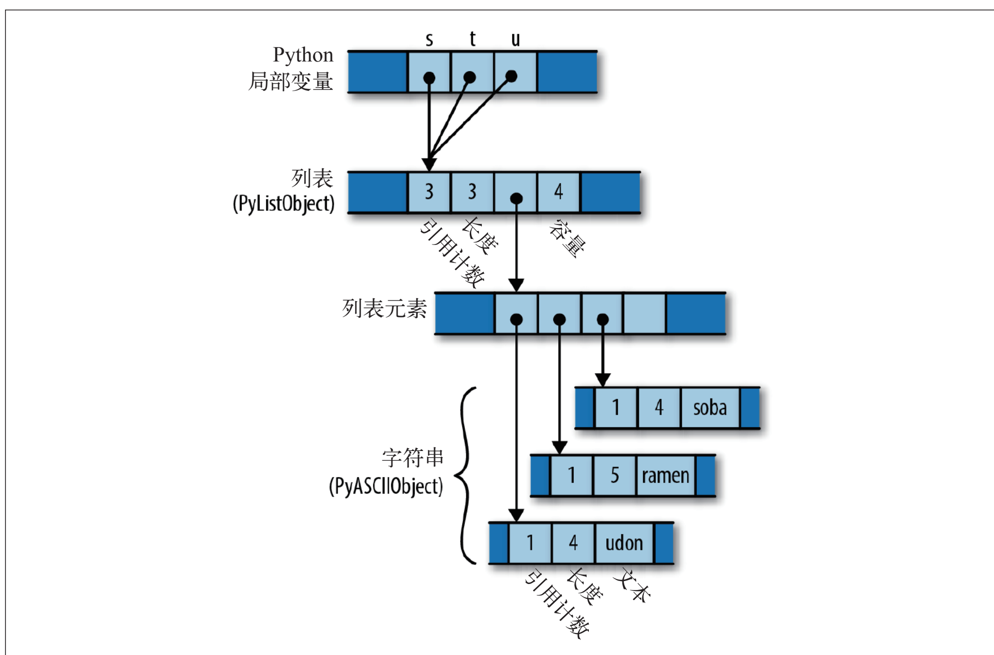


图 4-6：在 Python 中把 s 赋值给 t 和 u 之后的结果

Python 把指针从 s 复制到了 t 和 u，并把列表的引用计数更新为 3。Python 的赋值代价很小，因为它创建了一个指向对象的新引用，但同时又必须维护引用计数，以便知道何时可以释放这个值。

接下来再看看类似的 C++ 代码：

```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```

图 4-7 展示了 s 的原始值在内存中的样子。

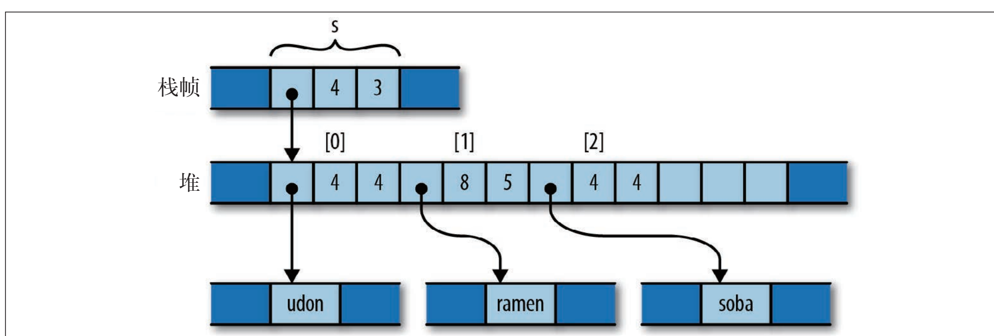


图4-7：C++字符串向量在内存中的表示

当程序将 s 赋值给 t 和 u 时会发生什么？ C++ 中对 std::vector 赋值会产生该向量的副本，std::string 的赋值也类似。因此当上面的代码执行之后，内存中会出现 3 个向量、9 个字符串，如图 4-8 所示。

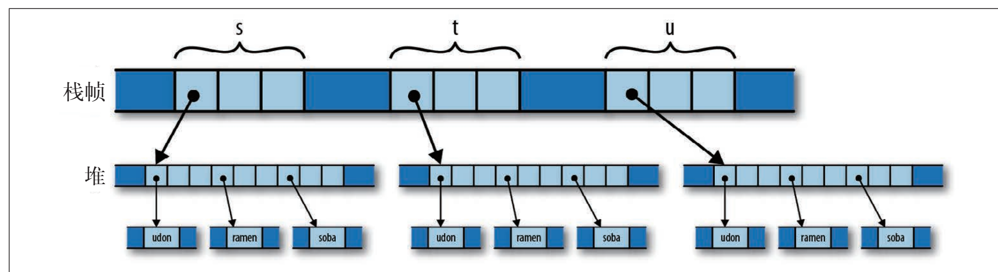


图 4-8：在 C++ 中把 s 赋值给 t 和 u 之后的结果

根据值的类型和大小，C++ 中的赋值可能会消耗较多内存和处理器时间。然而，这样实现的优点在于程序容易决定何时释放这些内存：只要变量超出作用域，在此分配的空间都会自动被清理。

从某种意义上讲，C++ 和 Python 在此选择了相反的思路：Python 赋值代价小，引用计数（一般情况下是垃圾收集）相对麻烦；C++ 保证了内存所有权的清晰，代价是赋值要深度复制对象。C++ 程序员对这个选择经常不大认同，因为深复制消耗大，而且通常都有更实用的替代方案。

那么，类似的程序在 Rust 中是什么样的？来看以下代码：

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
let t = s;
let u = s;
```

与 C 和 C++ 类似，Rust 会把 "udon" 这种纯字符串字面量放到只读内存中。为了与前面 C++ 和 Python 的例子作比较，这里调用 to_string 得到了分配在堆内存上的 String 值。

由于 Rust 对向量和字符串的表示与 C++ 类似，因此初始化 s 之后的状态看起来也跟 C++ 一样，如图 4-9 所示。

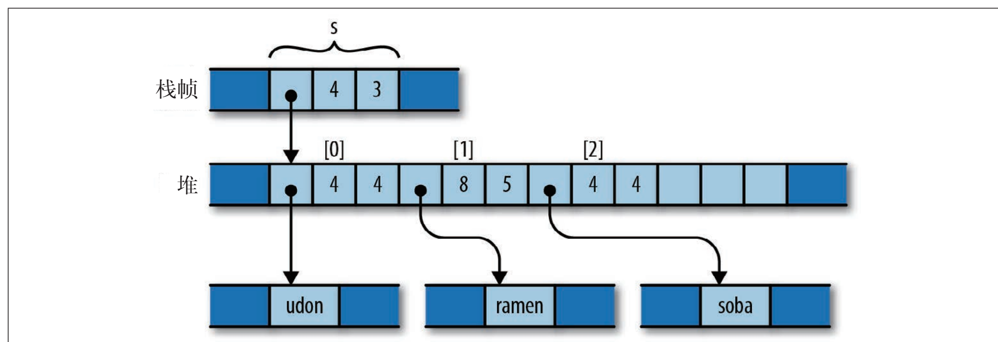


图4-9：Rust 字符串向量在内存中的表示

前面说过，Rust 中大多数类型的赋值是把值从源变量**转移**到目标变量，然后源变量变成未初始化状态。因此，初始化 `t` 之后，程序的内存如图 4-10 所示。

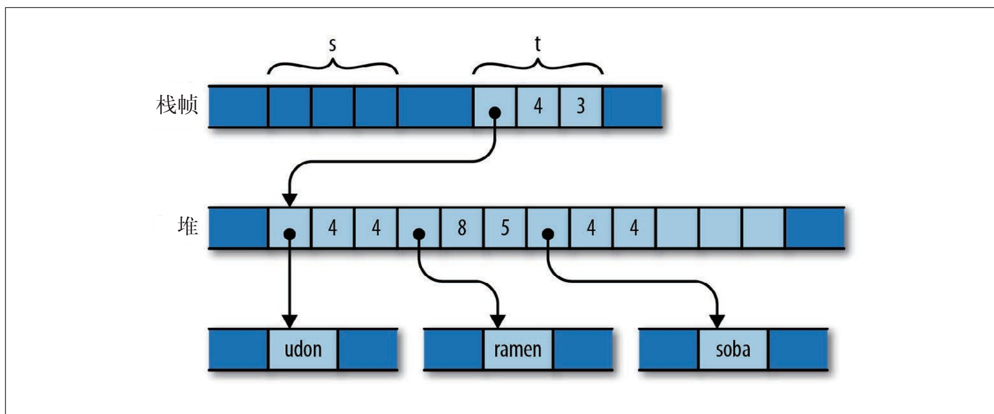


图 4-10：在 Rust 中把 s 赋值给 t 之后的结果

有什么不同？初始化代码 `let t = s;` 把向量的 3 个头部字段从 `s` 转移到了 `t`，然后 `t` 就拥有了这个向量。这个向量的元素仍然待在原处，字符串也没动。每个值依旧只有一个所有者，只不过转了一手。此时也没有引用计数需要调整。而编译器现在会认为 `s` 尚未初始化。

那么执行下一行初始化代码 `let u = s;` 时会发生什么？会把未初始化的值 `s` 赋给 `u`。Rust 对使用未初始化的值非常慎重，因此编译器拒绝执行这行代码并报出以下错误：

```
error[E0382]: use of moved value: `s`
 --> ownership_double_move.rs:9:9
  |
8 |     let t = s;
  |         - value moved here
9 |     let u = s;
  |         ^ value used here after move
  |
```

总结一下 Rust 使用转移之后的结果。与 Python 类似，赋值代价很小：程序仅仅把 3 个字的向量头部从一个地方转移到了另一个地方。而且也与 C++ 类似，所有者始终很清晰：程序不需要引用计数或垃圾收集，就可以知道什么时候释放向量元素及其字符串内容。

而代价呢？就是如果你需要这些值的副本，必须明确生成。假如你最终想要得到跟 C++ 程序一样的状态，即让每个变量都单独持有各自独立的结构，就必须调用向量的 `clone` 方法，这个方法会对向量及其元素执行深复制：

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];
let t = s.clone();
let u = s.clone();
```

当然，也可以再现 Python 的行为，只要使用 Rust 的引用计数的指针类型即可，稍后 4.4 节将讨论。

4.2.1 更多转移操作

前面的例子只是展示了变量初始化，即在进入作用域时通过 `let` 语句给变量提供一个值。而给变量赋值还有点不一样，体现在向已经初始化的变量转移值的时候。这时候，Rust 会清除这个变量之前的值。比如：

```
let mut s = "Govinda".to_string();
s = "Siddhartha".to_string(); // 值"Govinda"在这里被清除
```

这里，程序在将字符串 `"Siddhartha"` 赋值给 `s` 时，它之前的值 `"Govinda"` 会先被清除。再看这个例子：

```
let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string(); // 这里没有值被清除
```

这一次，`t` 取得了原来 `s` 的字符串的所有权。因此接下来给 `s` 赋值时，`s` 是未初始化的。在这种情况下，就没有字符串被清除了。

目前为止，我们只举了初始化和赋值的例子，因为这两个操作简单。但在 Rust 中，几乎任何使用值的地方，都适用于转移这个语义。给函数传递参数会将所有权转移给函数的参数，从函数返回值会将所有权转移给调用代码，构建元组也会把值转移到元组中，等等。

明白了这些，应该就更容易理解本章第一节的示例代码了。比如，在构建音乐家（composer）向量时，代码是这样写的：

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
```

这几行代码中就有几个地方涉及转移，当然不包括初始化和赋值。

- **从函数返回值**

  调用 `Vec::new()` 构建一个新向量并返回，注意不是指向这个向量的指针，而是向量本身。换句话说，新向量的所有权会从 `Vec::new` 转移到变量 `composers`。类似地，调用 `to_string` 方法也会返回一个全新的 `String` 实例。

- **构建新值**

  新 `Person` 结构体的 `name` 字段会以 `to_string` 返回的值初始化。这个结构体取得新字符串的所有权。

- **给函数传值**

  整个 `Person` 结构体，而不止是一个指针，被传递给了向量的 `push` 方法，结果是把结构体转移到了向量的末尾。向量取得了 `Person` 的所有权，因而也就成了 `name` 字段 `String` 的间接所有者。

像这样把值转移来转移去听起来似乎效率不高，但要记住两点。首先，转移的并不是值所拥有的堆内存上的内容，而只是**特征值**（value proper）。对向量和字符串来说，特征值是

只有 3 个字的头部，相对来说大得多的元素数组和文本缓冲区仍然待在它们在堆里原来的位置。其次，Rust 编译器在生成代码时能够“看穿”所有这些转移；实践中，这意味着在机器码中值通常会直接保存在拥有它的变量旁边。

4.2.2 转移与控制流

前面例子中的控制流都非常简单，在复杂代码中转移的语义是如何实现的呢？基本的原则就是，如果一个变量的值已经被转移了，而且转移后始终未得到新值，那么这个变量就被认为是未初始化的。比如，一个变量必须在 `if` 语句求值前后始终有值，才可以在两个分支中同时使用它：

```
let x = vec![10, 20, 30];
if c {
    f(x); //  ……在这里转移x的值没问题
} else {
    g(x); // ……在这里转移x的值也没问题
}
h(x) // 不行：任何一个分支转移x后，x在这里就变成未初始化了
```

出于类似的原因，循环体内的转移也是被禁止的：

```
let x = vec![10, 20, 30];
while f() {
    g(x); // 不行：x的值在第一次迭代时就被转移了，
          // 第二次迭代时它就变成未初始化了
}
```

换句话说，除非在下次迭代前给它一个新值：

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);           // x的值被转移了
    x = h();        // 给x一个新值
}
e(x);
```

4.2.3 转移与索引内容

前面提到转移会导致源变量变成未初始化状态，因为目标变量会取得原始值的所有权。然而，并不是任何类型的值都可以随意变成未初始化状态。比如，来看如下代码：

```
// 构建一个字符串向量："101"、"102"……"105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 从向量中取得随机值
let third = v[2];
let fifth = v[4];
```

如果让以上代码运行，Rust 则必须记住向量的第三个和第五个元素已经变成未初始化了，

而且这个信息要一直记录到向量被清除。最常见的处理方式是让向量本身携带额外的信息，用于记录哪个元素可以用，哪个元素则已经变成未初始化了。但对于一门系统编程语言来说，这显然不是正确的做法。向量就是向量，不应该携带多余的信息。事实上，Rust 在运行前面的代码时会给出如下错误：

```
error[E0507]: cannot move out of indexed content
  --> ownership_move_out_of_vector.rs:14:17
   |
14 |     let third = v[2];
   |                 ^^^^
   |                 |
   |                 help: consider using a reference instead `&v[2]`
   |                 cannot move out of indexed content
```

对于 fifth 也一样。在上面的错误消息中，如果你只想访问元素而不想转移它的值，那么 Rust 建议使用引用。一般来说，我们确实只想访问元素而不想转移值。但如果真的想要转移向量元素的值怎么办？那就需要采用规避这种类型限制的方法。以下是 3 个可能的方案：

```
// 构建一个字符串向量："101"、"102"……"105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 1. 从向量末尾取值：
let fifth = v.pop().unwrap();
assert_eq!(fifth, "105");

// 2. 从向量中间取值，同时用最后一个元素填充：
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 3. 用其他值来交换要取出的值：
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// 看看向量还剩什么内容吧
assert_eq!(v, vec!["101", "104", "substitute"]);
```

上面的方法都可以从向量中转移出值来，而且每种方法都能保证向量是被填满的状态，变小则没关系。

类似 Vec 这样的集合类型一般会提供方法，方便在循环中使用其所有元素：

```
let v = vec!["liberté".to_string(),
             "égalité".to_string(),
             "fraternité".to_string()];
for mut s in v {
    s.push('!');
    println!("{}", s);
}
```

在把向量直接传递给循环时，比如这里的 `for ... in v`，会把向量从 v 中**转移**出来，导致 v 变成未初始化。而 for 循环内部的机制会取得向量的所有权，并逐个迭代其元素。每次迭代，循环都会将一个元素转移到变量 s 中。因为 s 拥有当前的字符串，所以循环体内的代码可以在打印之前修改它。而由于向量本身对代码不再可见，因此循环期间无法观察到它半空的状态。

如果你发现要从中转移出值的所有者是编译器无法跟踪的，可以考虑改变所有者的类型，让自己可以动态跟踪它是否还有值。比如，以下是前面的一个示例的另一种写法：

```
struct Person { name: Option<String>, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: Some("Palestrina".to_string()),
                        birth: 1525 });
```

不能这样做：

```
let first_name = composers[0].name;
```

因为会引起跟前面一样的“cannot move out of indexed content”（不能转移索引内容）错误。但是，因为这里把 name 字段的类型由 String 改成了 Option<String>，所以也就意味着 None 是该字段可以拥有的一个有效值。因此，可以这样做：

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

这里 replace 调用转移出了 composers[0].name 的值，并把 None 补充回去，然后将原始值的所有权传递给了其调用者。事实上，像这样使用 Option 类型是非常常见的，于是该类型干脆专门为此提供了一个名为 take 的方法。因此实现前面的操作可以像下面这样把代码写得更简洁：

```
let first_name = composers[0].name.take();
```

这里调用 take 的结果跟前面调用 replace 一样。

4.3 Copy类型：转移的例外

目前为止本书所举例子中涉及的值包括向量、字符串和其他可能占用较多内存且复制起来比较耗时的类型。转移让这些类型的所有者保持清晰，赋值代价也小。但对于整数或字符这些比较简单的类型，这种小心翼翼的处理就真的没必要了。

下面比较一下 String 赋值与 i32 赋值在内存中的差异：

```
let str1 = "somnambulance".to_string();
let str2 = str1;

let num1: i32 = 36;
let num2 = num1;
```

运行以上代码后，内存如图 4-11 所示。

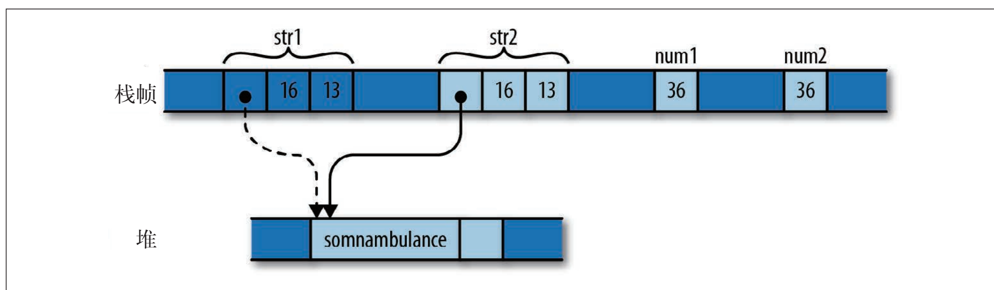


图 4-11：字符串赋值会转移值，i32 赋值则会复制值

与前面的向量一样，赋值把 str1 **转移**到 str2，因此不会出现两个字符串释放相同的缓冲区的情况。但 num1 和 num2 就不一样了。i32 只是内存中的一种位模式，它不拥有任何堆中的资源，或者说除了所包含的字节外，它什么也没有。那么在向 num2 转移其位时，就把 num1 完整复制了一份。

转移值会导致源变量变成未初始化。虽然将 str1 视为无值，将 num1 视为无意义的确很重要，但继续使用它没什么坏处。对数值来说，转移的好处不仅无法体现，反而会导致不便。

前面我们也谨慎地说过，**大多数**类型会转移。那么现在就来说说例外，即 Rust 选定为 Copy 类型。Copy 类型的赋值会复制值，而不是转移值。赋值的源变量仍然是初始化且可用的，值也跟先前一样。将 Copy 类型的值传递给函数和构造函数也类似。

标准的 Copy 类型包括所有机器整数和浮点数值类型、char 和 bool 类型，以及其他几种类型。Copy 类型的元组或固定大小的数组本身也是 Copy 类型。

只有可以简单地位到位复制的类型才是 Copy 类型。正如前面解释的，String 不是 Copy 类型，因为它拥有分配在堆上的缓冲区。基于同样的原因，Box<T> 也不是 Copy 类型，因为它拥有自己分配在堆上的引用值。表示操作系统文件勾柄的 File 类型也不是 Copy 类型，复制这种类型的值等于让操作系统再打开一个文件勾柄。类似地，表示加锁后的互斥量的 MutexGuard 类型也不是 Copy 类型：复制这种类型没什么意义，因为一个线程一次只能有一个互斥量。

有一条经验规则，就是任何在值被清除后需要特殊处理的类型都不能是 Copy 类型。比如，Vec 需要释放其元素，File 需要关闭其文件勾柄，而 MutexGuard 需要解锁其互斥量。这些类型的位到位的复制，会导致编译器分不清哪个值应该对原始的资源负责。

那么用户自定义的类型呢？默认情况下，struct 和 enum 类型不是 Copy 类型：

```
struct Label { number: u32 }

fn print(l: Label) { println!("STAMP: {}", l.number); }

let l = Label { number: 3 };
print(l);
println!("My label number is: {}", l.number);
```

以上代码无法通过编译，Rust 会报告如下错误：

```
error[E0382]: use of moved value: `l.number`
  --> ownership_struct.rs:12:40
   |
11 |     print(l);
   |           - value moved here
12 |     println!("My label number is: {}", l.number);
   |                                        ^^^^^^^^ value used here after move
   |
   = note: move occurs because `l` has type `main::Label`, which does not
           implement the `Copy` trait
```

因为 Label 不是 Copy 类型，所以把这种类型的值传给 print 会把值的所有权转移给 print，而 print 会在返回之前将该值清除。但这样做有点傻，Label 其实就是一个带点装饰的 u32。没有理由把 l 传给 print 还要转移值。

但用户定义的类型默认属于非 Copy 类型。如果自定义结构体的所有字段本身都是 Copy 类型，那可以在定义上方添加 #[derive(Copy, Clone)] 属性把这个类型标注成 Copy 类型，比如：

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

这样一来，代码就可以编译通过了。对于并非所有字段都是 Copy 类型的结构体，就算加这个属性也不管用。编译以下代码：

```
#[derive(Copy, Clone)]
struct StringLabel { name: String }
```

会引发以下错误：

```
error[E0204]: the trait `Copy` may not be implemented for this type
 --> ownership_string_label.rs:7:10
  |
7 | #[derive(Copy, Clone)]
  |          ^^^^
8 | struct StringLabel { name: String }
  |                      ------------ this field does not implement `Copy`
```

为什么不让用户定义的类型自动就是 Copy 且默认符合条件呢？因为一个类型是不是 Copy，对于什么样的代码可以使用它有很大影响。Copy 类型更灵活，因为赋值及相关操作不会导致原始变量未初始化。但对于类型实现者来说，反之亦然：Copy 类型非常受限制，因为它能包含的类型很少，不像非 Copy 类型那样可以使用堆空间且拥有其他资源。因此把某种类型实现为 Copy 类型，对于实现者而言意味着庄严的承诺：若日后有必要将其改为非 Copy 类型，那用到它的代码，很多可能需要重写。

虽然 C++ 支持重载赋值操作符和定义特殊的复制、移动构造函数，但 Rust 并不允许这种自定义。在 Rust 中，所有转移都是字节对字节的浅复制，会导致源变量未初始化。复制也一样，只不过源变量会保持初始化。当然，这意味着 C++ 类能够提供 Rust 类型无法提供的方便接口，看起来普普通通的代码就可以隐式调整引用计数，将开销大的复制操作延后，或者使用其他复杂的实现技巧。

但对于一门语言来说，C++ 的这种灵活性会导致赋值、传参和函数返回值变得不好预测。比如，本章前面的例子展示了在 C++ 中将一个变量赋值给另一个可能会占用任意内存空间

和处理器时间。Rust 的一个原则就是，开销应该对程序员显而易见。基本的操作必须保持简单。潜在开销大的操作应该明确，比如在前面的例子中调用 `clone` 会对向量及其包含的字符串执行深复制。

本节从一个类型应该具有的特征角度大概地讨论了 `Copy` 和 `Clone`。而实际上，它们都是**特型**（trait）的例子。特型是 Rust 的开放式机制，用于根据可以对数据执行什么操作来对它们进行分类。第 11 章将介绍特型，第 13 章则专门介绍 `Copy` 和 `Clone`。

4.4　Rc和Arc：共享所有权

尽管典型的 Rust 代码中的大多数值拥有唯一的所有者，但在某些情况下很难找到每个值只有一个所有者时所需的生命期。你可能希望某个值在所有操作都完成时再被清除。对此，Rust 提供了基于引用计数的指针类型 `Rc` 和 `Arc`。正如 Rust 所承诺的，这两种类型可以安全使用。放心，你不会忘记调整引用计数，不会对 Rust 未关注的值创建指针，也不会因为 C++ 中那些与引用计数指针相关的问题而碰到麻烦。

`Rc` 和 `Arc` 类型非常相似，唯一的区别在于 `Arc`（`Arc` 是 atomic reference count，即**原子引用计数**的简写）可以在线程间安全共享，`Rc` 则会使用更快的非线程安全的代码更新其引用计数。如果不需要在线程间共享指针，则没必要承担 `Arc` 带来的性能开销，就用 `RC` 即可；Rust 会阻止跨线程传递它。除此之外，这两种类型是等价的，因此本节后面就只讨论 `Rc`。

本章前面展示了 Python 使用引用计数管理其值的生命期的例子。在 Rust 中，可以使用 `Rc` 达成类似的效果。来看如下代码：

```
use std::rc::Rc;

// Rust可以推断所有这些类型，这里写出来是为了清楚
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

对于任意类型 `T`，`Rc<T>` 值是一个指针，指向堆空间中的 `T` 值，同时该值有一个附属的引用计数。对 `Rc<T>` 调用 `clone` 方法不会复制 `T`，只会创建另一个指向它的指针并给引用计数加 1。因此，前面的代码执行后，内存会如图 4-12 所示。

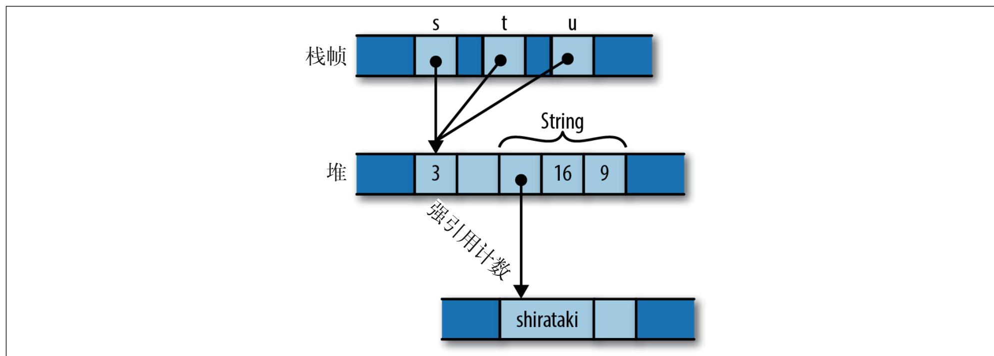


图 4-12：引用计数的字符串，有 3 个引用指向它

3 个 Rc<String> 指针引用的是同一块内存，其中包含一个引用计数和 String 的空间。通常的所有权规则适用于 Rc 指针本身，当最后一个 Rc 被清除后，Rust 也会清除对应的 String。

在 Rc<String> 上也可以直接使用 String 的所有常用方法：

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{} are quite chewy, almost bouncy, but lack flavor", u);
```

Rc 指针拥有的值不可修改。如果要在字符串末尾追加一些文本：

```
s.push_str(" noodles");
```

Rust 会拒绝：

```
error: cannot borrow immutable borrowed content as mutable
  --> ownership_rc_mutability.rs:12:5
   |
12 |     s.push_str(" noodles");
   |     ^ cannot borrow as mutable
```

Rust 的内存和线程安全保证有一个前提，即不存在既共享又可以修改的值。Rust 假设 Rc 指针引用的值通常是共享的，因此应该不可修改。第 5 章将解释为什么这个限制很重要。

使用引用计数管理内存的另一个众所周知的问题是，如果存在两个引用计数的值互相指向对方，则双方的引用计数将始终大于 0，因此这两个值将永远不会被释放（如图 4-13 所示）。

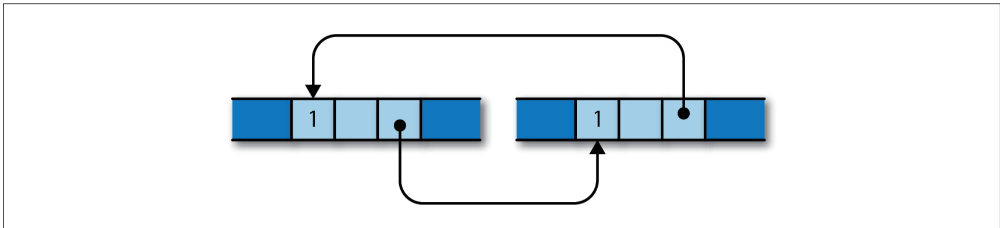


图 4-13：引用计数循环，这两个对象永远得不到释放

在 Rust 中，这样的循环引用可能导致内存泄漏，但很少见。因为要创建循环引用，必须在某个时刻让旧值指向新值。而这显然要求旧值必须可修改。由于 Rc 指针引用的值不可修改，因此正常情况是不可能创建循环的。不过，Rust 确实支持给不可修改的值创建可修改的部分，这个能力叫**内部修改能力**（interior mutability），9.9 节将介绍。如果同时使用该技术和 Rc 指针，是可以创造循环引用并导致内存泄漏的。

有时候，可以使用**弱指针**（weak pointer）std::rc::Weak 取代一些链接以避免创建 Rc 指针的循环。不过，本书不会涉及相关内容，具体细节可以参考标准库的文档。

转移和引用计数指针是两种缓和所有权树严苛局面的方式。下一章将介绍第三种方式：借用对某个值的引用。熟悉了所有权和借用的概念之后，才算是跨过了学习 Rust 的门槛，之后就可以专注于利用它的独特优势了。

第 5 章

引用

能力不足，怨不得书。

——Mark Miller

目前为止我们看到的所有指针类型，包括 `Box<T>` 堆指针以及 `String`、`Vec` 值内部的指针，都是所有型指针。这意味着当所有者被清除时，引用的资源也会随之清除。Rust 也有一种非所有型指针类型，叫作**引用**（reference），这种指针对它所引用资源的生命期没有影响。

事实上，恰恰相反：引用的生命期不能超过其引用的资源。换句话说，代码中引用的生存期不能超过它指向的值，必须让这一点在代码中显而易见。为强调这一点，Rust 把创建对某个值的引用称作**借用**（borrow）这个值：借的东西，迟早要还给它的所有者。

如果看到“必须让这一点在代码中显而易见”这句话你有所怀疑，这很正常。引用本身没什么特别的，说到底，引用就是地址。只不过 Rust 保证引用安全的规则是比较新奇的，纵观同类语言，应该说史无先例。虽然这些规则属于 Rust 中最难掌握的部分，但它们可以神奇地帮你避免各式各样、天天都会有的 bug，而且对于多线程编程也具有解放意义。同样，这又是 Rust 激进的赌注。

来看一个例子。假设我们想做一个包含文艺复兴时期艺术家及其知名作品的表。Rust 标准库有一个散列表类型，基于它可以创建我们的自定义类型：

```
use std::collections::HashMap;

type Table = HashMap<String, Vec<String>>;
```

换句话说，这是一个把 `String` 值映射到 `Vec<String>` 值的散列表，可以根据艺术家的名字找到对应的作品。可以使用 `for` 循环迭代 `HashMap`，因此，为了调试我们写一个函数来打印 `Table` 的内容：

```
fn show(table: Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

构建和打印这个表很简单：

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                 vec!["many madrigals".to_string(),
                      "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                 vec!["The Musicians".to_string(),
                      "The Calling of St. Matthew".to_string()]);
    table.insert("Cellini".to_string(),
                 vec!["Perseus with the head of Medusa".to_string(),
                      "a salt cellar".to_string()]);

    show(table);
}
```

运行也一切正常：

```
$ cargo run
     Running `/home/jimb/rust/book/fragments/target/debug/fragments`
works by Gesualdo:
  Tenebrae Responsoria
  many madrigals
works by Cellini:
  Perseus with the head of Medusa
  a salt cellar
works by Caravaggio:
  The Musicians
  The Calling of St. Matthew
$
```

但是，如果你看过前一章关于转移的讨论，那么应该会对 `show` 的定义产生几个疑问。特别地，`HashMap` 不是 `Copy`，因为它拥有动态分配的表。所以当程序执行到 `show(table)` 时，整个结构都会转移到函数中，变量 `table` 变成了未初始化。如果此时代码再尝试使用 `table`，就会出问题：

```
...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

Rust 提醒 `table` 的值已经不存在了：

```
error[E0382]: use of moved value: `table`
  --> references_show_moves_table.rs:29:16
```

```
   |
28 |     show(table);
   |          ----- value moved here
29 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
   |                ^^^^^ value used here after move
   |
   = note: move occurs because `table` has type `HashMap<String, Vec<String>>`,
           which does not implement the `Copy` trait
```

实际上，如果再看一看 show 的定义，会发现外层 for 循环取得了这个散列表的所有权并用尽了它。内层的 for 循环也一样，只不过使用的是各个字符串向量。（4.2.3 节的例子中也有过类似的情况。）因为存在转移的语义，所以这里打印完内容之后就把整个结构销毁了。Rust，真有你的！

这里正确的做法是使用引用。通过引用，可以访问值，又不会影响其所有权。引用分两种情况。

- **共享引用**（shared reference），可以读取引用的值，但不能修改。不过，你可以拥有对某个特定值的任意多个共享引用。表达式 &e 会产生一个对 e 的值的共享引用。如果 e 的类型是 T，那么 &e 的类型就是 &T，读作"T 的引用"。共享引用是 Copy。
- **可修改引用**（mutable reference），可以读取和修改引用的值。不过，你不能同时拥有对该值的任何其他引用。表达式 &mut e 会产生一个对 e 的值的可修改引用。如果把 e 的类型写作 &mut T，就可以读作"T 的可修改引用"。可修改引用不是 Copy。

可以将共享引用和可修改引用的区别理解为它们会在编译时分别执行**多读**（multiple readers）和**单写**（single writer）的检查规则。事实上，这条规则不仅仅适用于引用，也适用于被借用值的所有者。只要存在对某个值的共享引用，即便该值的所有者也不能修改它。此时的值被锁定了。比如，在 show 执行期间，没有人可以修改 table。类似地，如果存在对某个值的可修改引用，则该引用对这个值拥有排他读写权。换句话说，在这个可修改引用存续期间，连对应值的所有者你都无法使用。保持共享引用和可修改引用分开，是保障内存安全的基本前提，本章稍后会进一步解释。

我们例子中的打印函数不需要修改散列表，只需读取其内容。因此，调用时可以只传入对散列表的共享引用：

```
show(&table);
```

引用是非所有型指针，因此 table 变量仍然是整个结构的所有者，show 只是借用一下而已。自然，还需要相应地调整一下 show 的定义，不过你得仔细看才能看出差别来：

```
fn show(table: &Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

show 的参数 table 已经从 Table 变成了 &Table：以前传的是值（实际上是把所有权转移到

了函数中)，现在传的是共享引用。这是代码中唯一要改动的地方。那么函数体里不改也可以吗?

首先，原来的外层 for 循环在修改前取得了 HashMap 的所有权并使用了它，在修改后则得到了对 HashMap 的一个共享引用。迭代对 HashMap 的共享引用，按照定义会产生对其中每一项键和值的共享引用：artist 从原来的 String 变成了 &String，而 works 从原来的 Vec<String> 变成了 &Vec<String>。

内层循环也会发生类似的变化。迭代对向量的共享引用，按照定义会产生对其元素的共享引用，因此 work 现在是 &String。此时函数中没有发生所有权转手，只有非所有型的引用在传递。

现在，如果想写一个函数，按字母表顺序给每位艺术家的作品排个序，那共享引用就不行了，因为共享引用不允许修改。没错，这个排序函数需要取得散列表的可修改引用：

```
fn sort_works(table: &mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```

调用时要传入它：

```
sort_works(&mut table);
```

通过可修改借用，sort_works 获得了读取和修改结构的能力，对向量的 sort 方法来说这是必需的。

当以转移所有权的方式给函数传参时，我们称其为**传值**（by value）。如果传给函数的是对值的引用，则称其为**传引用**（by reference）。比如，我们修改了 show 函数，由原来给它传值改为了传引用。很多语言会强调这个差异，但这对 Rust 而言尤其重要，因为它体现了影响所有权的不同方式。

5.1 引用作为值

前面的例子展示了引用非常典型的使用场景，即让函数可以访问或操作某个结构，但又不必取得其所有权。而引用实际上比这还要灵活，本节将通过几个例子来说明这一点。

5.1.1 Rust引用与C++引用

如果你熟悉 C++ 引用，就会发现它们与 Rust 引用有一定的共通之处。最重要的是，它们在机器级别上都是地址。而在实践中，Rust 引用给人的感觉非常不一样。

在 C++ 中，引用按惯例是隐式创建的，解引用也是隐式的：

```
// C++代码!
int x = 10;
int &r = x;             // 初始化时隐式创建引用
assert(r == 10);        // 隐式地对r解引用以读取x的值
r = 20;                 // 把20保存在x中，r本身仍然指向x
```

在 Rust 中，引用是通过 & 操作符显式创建的，而解引用也要显式使用 * 操作符：

```
// 从这里开始都是Rust代码了
let x = 10;
let r = &x;             // &x是对x的共享引用
assert!(*r == 10);      // 显式地对r解引用
```

要创建可修改引用，则使用 &mut 操作符：

```
let mut y = 32;
let m = &mut y;       // &mut y是y的可修改引用
*m += 32;             // 显式地对m解引用以便设置y的值
assert!(*m == 64);    // 读取y的新值同样要显式解引用
```

等等，还记得吗，我们在修改 show 函数，给它传引用而不是传值之后，并没有使用 * 操作符来解引用。这是怎么回事？

由于 Rust 代码中会广泛使用引用，因此 . 操作符会在必要时对其左操作数进行隐式解引用：

```
struct Anime { name: &'static str, bechdel_pass: bool };
let aria = Anime { name: "Aria: The Animation", bechdel_pass: true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// 等价于上一行代码，只不过把解引用写明了：
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

前面 show 函数中的宏 println! 会扩展成使用 . 操作符的代码，因此也会利用该操作符的隐式解引用。

除了隐式解引用，. 操作符在方法调用需要时还可以隐式借用对其左操作数的引用。比如，Vec 的 sort 方法需要取得向量的可修改引用，此时下面两种调用方式是等价的：

```
let mut v = vec![1973, 1968];
v.sort();           // 隐式借用了对v的可修改引用
(&mut v).sort();    // 等价，更难看
```

总结一下，C++ 会隐式地在引用和左值（即引用内存位置的表达式）之间转换，这些转换会出现在任何需要用到它们的地方。而在 Rust 中，则必须使用 & 和 * 操作符来创建和追随引用，只有 . 操作符例外，它会隐式地借用和解引用。

5.1.2　给引用赋值

给 Rust 引用赋值会导致它指向新值：

```
let x = 10;
let y = 20;
let mut r = &x;

if b { r = &y; }

assert!(*r == 10 || *r == 20);
```

引用 r 开始指向 x，而如果 b 为 true，它就会指向 y，如图 5-1 所示。

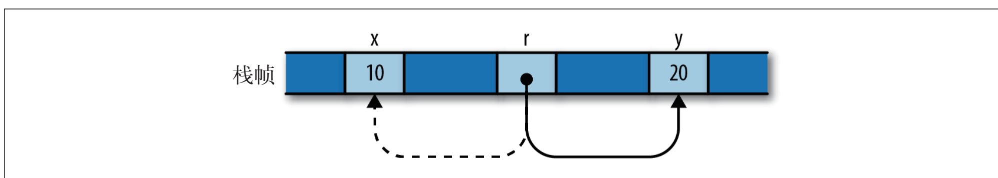


图 5-1：引用 r 现在指向了 y 而不是 x

这跟 C++ 完全不同，在 C++ 中给引用赋值会将值存储在引用中。而且，没有办法将 C++ 引用指向初始值之外的其他地址。

5.1.3 引用的引用

Rust 允许使用引用的引用：

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr: &&&Point = &rr;
```

（这里明确写出了引用的类型，但实际上是可以省略的，Rust 可以自行推断。）不管引用了多少个引用，. 操作符都可以顺藤摸瓜地找到最终的值：

```
assert_eq!(rrr.y, 729);
```

在内存中，这几个引用如图 5-2 所示。

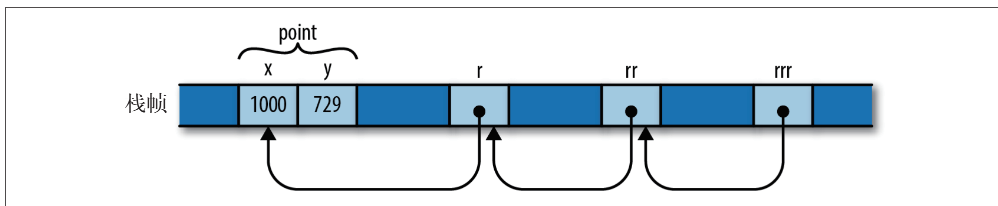


图 5-2：引用的引用的链条

表达式 `rrr.y` 按照 `rrr` 类型的指示，会跨越 3 层引用找到 `Point` 并取得其 `y` 字段的值。

5.1.4 比较引用

与 . 操作符类似，Rust 的比较操作符也能“看穿”任意多个引用，只要两个操作数的类型相同即可：

```
let x = 10;
let y = 10;

let rx = &x;
let ry = &y;
```

```
let rrx = &rx;
let rry = &ry;

assert!(rrx <= rry);
assert!(rrx == rry);
```

最后一个断言会成功，即使 rrx 和 rry 指向不同的值（即 rx 和 ry），因为 == 操作符会跟随所有引用，对它们的最终目标 x 和 y 进行比较。基本上这正是我们所希望的，特别是在编写泛型函数时。如果确实想比较两个引用是不是指向同一块内存，可以使用 std::ptr::eq，这个方法比较引用的地址：

```
assert!(rx == ry);              // 它们引用的值相等
assert!(!std::ptr::eq(rx, ry)); // 但两个值不在一个地址
```

5.1.5　引用永远不为空

Rust 引用永远不为空。没有跟 C 的 NULL 或 C++ 的 nullptr 对应的东西存在。引用没有默认的初始值（无论什么类型的变量，在其初始化之前都不能使用）。而且，Rust（在 unsafe 代码外部）不会将整数转换为引用，因此不能把 0 转换成引用。

C 和 C++ 代码经常使用空指针表示某个值不存在。比如，malloc 函数可能返回指向一块新内存的指针，也可能在内存不足时返回 nullptr。而在 Rust 中，如果你需要一个要么是引用要么什么也不是的值，那就使用 Option<&T>。在机器级别，Rust 将 None 表示为空指针，将 Some(r)（其中 r 是 &T 值）表示为非零地址。因此 Option<&T> 完全可以像 C 或 C++ 中的空指针一样有效，但更安全：它的类型要求你在使用之前必须检查它是否为 None。

5.1.6　借用对任意表达式的引用

与 C 和 C++ 只允许对某些类型的表达式应用 & 操作符不同，Rust 允许你借用对任何类型表达式的值的引用：

```
fn factorial(n: usize) -> usize {
    (1..n+1).fold(1, |a, b| a * b)
}
let r = &factorial(6);
assert_eq!(r + &1009, 1729);
```

在类似这种情况下，Rust 会创建一个匿名变量来保存表达式的值，然后生成一个指向该值的引用。这个匿名变量的生命期取决于你会对引用做什么。

- 如果你在 let 语句中立即把这个引用赋给一个变量（或将其变成立即被赋值的结构体或数组的一部分），那 Rust 会让这个匿名变量具有与 let 初始化的变量一样长的生命期。在前面的例子中，r 会一直引用对应的匿名变量。
- 否则，匿名变量会存活至闭合语句的末尾。对我们的例子而言，用于保存 1009 的匿名变量只会存活至 assert_eq! 语句结束。

如果你对 C 或 C++ 比较了解，可能会觉得这样容易出问题。但是不要忘了，Rust 永远不会让你的代码中出现悬空指针。如果有超出匿名变量生命期且永远不会用到的引用存在，

Rust 一定会在编译时发现并报告它。你只要把匿名变量的值保存到一个命名变量中并给它一个适当的生命期即可。

5.1.7　对切片和特型对象的引用

目前为止我们看到的引用全部是简单的地址。然而，Rust 也有两种**胖指针**（fat pointer），即包含某个值的地址以及与使用该值相关的必要信息的一个两个字的值。

对切片的引用是一个胖指针，包含切片地址及其长度信息。第 3 章详细介绍过切片。

Rust 的另一种胖指针是特型对象（trait object），即对实现某种特型的一个值的引用。特型对象包含一个值的地址和一个指向与该值匹配的特型实现的指针，以便于调用特型的方法。11.1.1 节将详细介绍特型对象。

除了携带这些额外的数据，切片和特型对象的引用与本章的其他引用没什么不同：它们都不拥有自己指向的值，它们的生命期都不能超出目标值，它们可以是可修改的或共享的，等等。

5.2　引用安全

正如你所看到的，引用与 C 或 C++ 中的普通指针非常相似。但这种指针是不安全的，Rust 是怎么控制引用的呢？解释规则的最好方式或许就是打破规则。我们先从最简单的例子开始，然后逐步增加复杂性并解释原因。

5.2.1　借用局部变量

下面是一个非常明显的例子。不能借用对一个局部变量的引用，然后将其拿到该变量的作用域之外：

```
{
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
}
```

Rust 编译器拒绝了这段程序，并给出了详细的错误消息：

```
error: `x` does not live long enough
  --> references_dangling.rs:8:5
   |
7  |         r = &x;
   |              - borrow occurs here
8  |     }
   |     ^ `x` dropped here while still borrowed
9  |     assert_eq!(*r, 1);  // bad: reads memory `x` used to occupy
10 | }
   | - borrowed value needs to live until here
```

Rust 抱怨的是 x 只存活到内部块的末尾，引用却存活到了外部块的末尾，变成了悬空指针，这是不允许的。

虽然这段程序无法执行的原因对人类而言显而易见，但还是有必要看一看 Rust 是怎么得出这个结论的。即便是这么简单的例子，也可以帮我们理解 Rust 检查更复杂代码时所用到的逻辑工具。

Rust 会给程序中的每个引用类型附加一个**生命期**（lifetime），生命期的长短与如何使用该引用匹配。生命期是程序中可以安全使用引用的一个范围，比如一个词法块、一个语句、一个表达式、某个变量的作用域，等等。生命期完全是 Rust 在编译时虚构的东西。而在运行时，引用就是一个地址，其生命期取决于自身的类型，没有运行时表示。

在上面的例子中，有 3 个生命期的关系需要搞清楚。变量 r 和 x 都有生命期，始于它们被初始化，止于离开它们的作用域。第三个生命期是一个引用类型的生命期，即借用 &x 并保存在 r 中的引用的生命期。

下面是一个看起来显而易见的约束：对于变量 x 的引用不能比 x 本身还“长寿”，如图 5-3 所示。

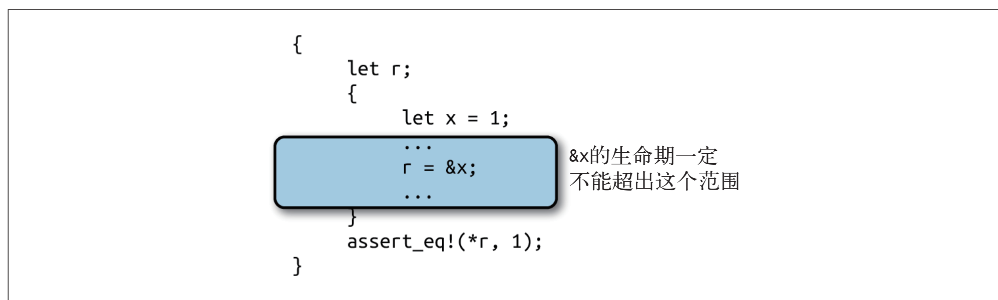


图 5-3：&x 可能的生命期

离开 x 的作用域之后，对它的引用会变成悬空指针。为此，我们说变量的生命期必须**包含**或**涵盖**从它那里借来的引用的生命期。

下面再看另一个约束：保存在变量 r 中的引用，其类型必须保证它在变量的整个生命期都有效，自初始化始，至离开作用域止，如图 5-4 所示。

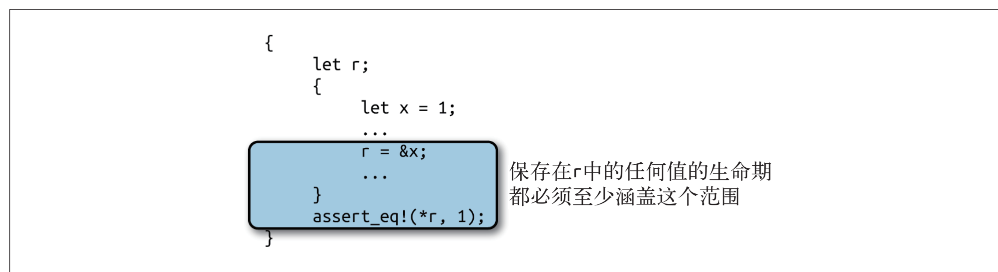


图 5-4：保存在 r 中的引用的容许生命期

如果引用不能至少与保存它的变量一样“长寿”，那么 r 在某个时刻就会成为悬空指针。为此，我们说引用的生命期必须包含或涵盖保存它的变量的生命期。

上述第一个约束限制引用的生命期可以有多长，而第二个约束限制引用的生命期可以有多短。Rust 只要找到一个同时满足这两个约束的生命期即可。可是，在我们的例子中没有这样一个生命期，如图 5-5 所示。

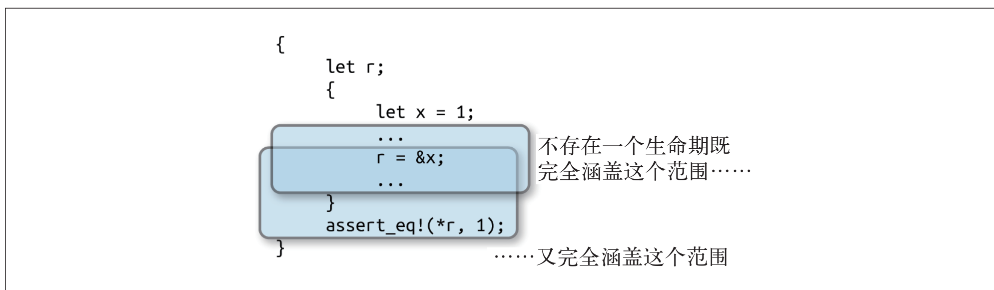


图 5-5：引用的生命期约束存在冲突

现在来看一个可以满足上述约束的不同的例子吧。约束还跟之前一样：引用的生命期必须包含在 x 的生命期内，同时必须完全涵盖 r 的生命期。由于 r 的生命期现在变短了，因此就出现了一个满足约束的生命期，如图 5-6 所示。

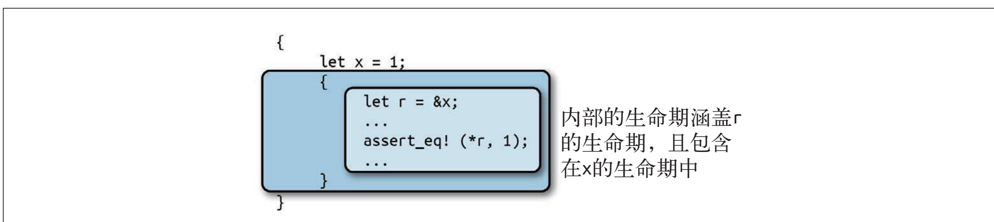


图 5-6：生命期包含 r 的作用域，同时也在 x 的作用域中的引用

在从较大数据结构中借用某个部分的引用时，这些规则会非常自然地应用，比如借用向量中的元素：

```
let v = vec![1, 2, 3];
let r = &v[1];
```

由于 v 拥有向量，而向量拥有自己的元素，因此 v 的生命期必须涵盖引用类型 &v[1] 的生命期。类似地，如果把引用保存在某个数据结构中，其生命期必须涵盖该数据结构的生命期。比如，对于一个引用向量而言，（作为向量元素的）所有引用的生命期都必须涵盖拥有这个向量的变量的生命期。

这是 Rust 评判所有代码的流程的本质。无论给语言增加什么新特性，比如数据结构或函数调用，都会相应引入新的约束条件。但原理始终不变：首先，理解由程序使用引用的方式带来的约束；其次，找到满足约束的生命期。这跟 C 和 C++ 程序员强加给自己的流程没有太大区别；区别仅在于 Rust 知道这些规则，并会强制代码遵守。

5.2.2 接收引用作为参数

在把引用传给函数时，Rust 如何确保函数能安全地使用它？假设函数 `f` 取得了一个引用并将其保存在了一个全局变量中。整个过程会修改几次，先看第一稿：

```
// 这段代码有些问题，过不了编译
static mut STASH: &i32;
fn f(p: &i32) { STASH = p; }
```

Rust 中与全局变量等价的是 `static` 变量：这是一种在程序启动时创建并一直存活到程序终止时的值。（跟其他声明一样，Rust 的模块系统控制静态变量在哪里可见，因此这里的"全局"指的是它们的生命期，而不是可见性。）第 8 章会介绍静态变量，此处只讨论前面的代码违背的几条规则。

- 所有静态变量都必须初始化。
- 可修改静态变量本质上不是线程安全的（毕竟，任何线程在任意时刻都可能访问静态变量），即使在单线程的程序中，它们也可能成为其他常见问题的牺牲品。为此，建议只在 `unsafe` 块中存取可修改静态变量。在这个例子中，我们不关心那些特殊问题，因此可以简单写个 `unsafe` 块，然后继续。

修改之后的代码如下：

```
static mut STASH: &i32 = &128;
fn f(p: &i32) { // 还不够好
    unsafe {
        STASH = p;
    }
}
```

这就差不多了。要发现剩下的问题，必须把 Rust 允许我们省略的一些代码也写出来。函数 `f` 的签名如果完整地写出来，是这样的：

```
fn f<'a>(p: &'a i32) { ... }
```

这里，生命期 `'a`（读作"撇 A"）是 `f` 的**生命期参数**（lifetime parameter），其中 `<'a>` 相当于"对于任意生命期 `'a`"，因此 `fn f<'a>(p: &'a i32)` 表示这个函数接收一个具有任意给定生命期 `'a` 的 `i32` 的引用。

由于必须允许 `'a` 为任意生命期，因此如果能让它的生命期尽可能最短，即恰好包含对 `f` 的调用，事情就会比较好办。此时赋值就成了焦点：

```
STASH = p;
```

因为 `STASH` 的生命期与程序的整个执行过程一样长，所以它保存的引用类型的生命期也必须同样长才行。Rust 称这种生命期为 `'static` 生命期。但 `p` 的生命期是 `'a`，即任意长度的生命期，只要能涵盖对 `f` 的调用就行。因此，Rust 拒绝编译这段代码：

```
error[E0312]: lifetime of reference outlives lifetime of borrowed content...
 --> references_static.rs:6:17
  |
6 |         STASH = p;
  |                 ^
```

```
  |
  = note: ...the reference is valid for the static lifetime...
note: ...but the borrowed content is only valid for the anonymous lifetime #1
      defined on the function body at 4:0
 --> references_static.rs:4:1
  |
4 | / fn f(p: &i32) { // 还不够好
5 | |     unsafe {
6 | |         STASH = p;
7 | |     }
8 | | }
  | |_^
```

此时，很明显我们的函数不能接受任意引用作为参数，但应该能够接受一个具有 'static 生命期的引用。只有把这样一个引用保存在 STASH 中才不会创建悬空指针。确实，下面的代码编译就通过了：

```
static mut STASH: &i32 = &10;

fn f(p: &'static i32) {
    unsafe {
        STASH = p;
    }
}
```

这次，f 的签名中加上了 p 必须是一个具有 'static 生命期引用的限制，因此把它保存到 STASH 中就不会出问题了。当然，这样一改就只能对那些静态变量的引用使用 f 了，毕竟只有静态变量的引用才不会导致 STASH 成为悬空指针。比如，可以这样写：

```
static WORTH_POINTING_AT: i32 = 1000;
f(&WORTH_POINTING_AT);
```

既然 WORTH_POINTING_AT 是静态的，&WORTH_POINTING_AT 的类型是 &'static i32，那把它传给 f 就是安全的。

现在回头看看在逐步修正代码的过程中，f 的签名有哪些变化：最初是 f(p: i32)，最终是 f(p: &'static i32)。换句话说，在 Rust 中不可能写出与函数签名意图不匹配的函数来，比如在最初的写法下想把引用保存到全局变量中是不可能的。Rust 中函数的签名始终反映函数体的行为。

相反，如果真看到一个函数的签名是 g(p: &i32)（或者把生命期也写出来，即 g<'a>(p: &'a i32)），那马上就可以知道这个函数**不能**把参数 p 保存到生命期超出调用之外的变量里。根本不用看 g 的定义，只看签名就知道 g 对自己的参数能做什么，不能做什么。这个事实正是确保函数调用安全的一个基础。

5.2.3　将引用作为参数传递

前面讨论了函数签名与函数体的关系，接下来看看函数与调用者的关系。假设有以下代码：

```
// 这个函数可以简写成：fn g(p: &i32)，
// 不过，现在先把生命期也写上
```

```
fn g<'a>(p: &'a i32) { ... }

let x = 10;
g(&x);
```

单从 g 的签名看，Rust 知道它不会把 p 保存到超出调用生命期的变量里：任何涵盖调用的生命期都满足 'a。因此 Rust 在此为 &x 选择了尽可能短的生命期：对 g 的调用。这个生命期满足所有约束：没有超出 x 且涵盖对 g 的全部调用。因此代码顺利编译通过。

注意，尽管 g 的定义中包含生命期参数 'a，但调用 g 时并不用管它。一般来说，只需在定义函数和类型时考虑生命期参数，而在使用它们时 Rust 会为你推断生命期。

如果把这里的 &x 传给前面定义的把自己的参数保存在静态变量里的函数 f，会怎么样?

```
fn f(p: &'static i32) { ... }

let x = 10;
f(&x);
```

编译失败：引用 &x 不能“活”得比 x 还长，而把它传给 f，会强迫它“活”得跟 'static 一样长。这里没办法做到“人人满意”，Rust 只能拒绝编译。

5.2.4 返回引用

函数中取得某个数据结构的引用，然后返回对该结构某一部分的引用，这是很常见的。比如，下面的函数会返回一个对切片中最小元素的引用：

```
// v至少包含一个元素
fn smallest(v: &[i32]) -> &i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

这里像平常一样简写了函数的签名。如果一个函数接收一个引用作为参数并返回一个引用，那么 Rust 会假设这两个引用具有相同的生命期。完整地写出来就是：

```
fn smallest<'a>(v: &'a [i32]) -> &'a i32 { ... }
```

假设这样调用 smallest：

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // 不行：指向了已被清除的数组的元素
```

根据 smallest 的签名可知其参数和返回值必须具有相同的生命期 'a。在上面的调用中，参数 &parabola 不能“活”得比 parabola 还长，可是 smallest 的返回值至少要跟 s 一样“长寿”。此时不存在能同时满足两个约束的生命期 'a，因此 Rust 拒绝编译：

```
error: `parabola` does not live long enough
  --> references_lifetimes_propagated.rs:12:5
   |
11 |         s = smallest(&parabola);
   |                       -------- borrow occurs here
12 |     }
   |     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // 不行：指向了已被清除的数组的元素
14 | }
   | - borrowed value needs to live until here
```

把 s 挪一下，让它的生命期明显包含在 parabola 的生命期中，问题就解决了：

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // 可以：parabola还活着
}
```

函数签名中的生命期让 Rust 可以评估传递给函数的引用和函数返回的引用之间的关系，从而确保安全地使用它们。

5.2.5 结构体包含引用

Rust 如何处理保存在数据结构中的引用呢？下面的程序跟前面一样也有错误，只不过结构体中包含了引用：

```
// 这段代码不能通过编译
struct S {
    r: &i32
}

let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10); // 不行：从已被清除的x中读取
```

Rust 对引用施加的安全约束不会因为把引用藏在了结构体中而失效。实际上，这些约束最终也会应用给 S。确实，Rust 质疑了：

```
error[E0106]: missing lifetime specifier
 --> references_in_struct.rs:7:12
  |
7 |         r: &i32
  |            ^ expected lifetime parameter
```

当引用类型出现在另一个类型的定义中时，必须写出其生命期。比如，可以这样写：

```
struct S {
    r: &'static i32
}
```

这意味着 r 只能引用生命期与程序一样长的 i32 值，而这个限制太大了。为此，可以为这个类型指定一个生命期参数 'a，并让它作用于 r：

```
struct S<'a> {
    r: &'a i32
}
```

S 类型现在有了一个生命期，就像引用类型一样。此后，每个类型 S 的值都会有一个新的生命期 'a，它会限制你使用这个值的方式。保存在 r 中的任何引用的生命期最好包含 'a，而 'a 也必须比保存 S 的任何值都“长寿”。

回到前面的代码，表达式 S { r: &x } 创建了一个带有生命期 'a 的新 S 值。在把 &x 保存在 r 字段中时，等于把 'a 完全限制在了 x 的生命期内。

赋值语句 s = S { ... } 把这个 S 保存在了一个生命期直到程序结束的变量中，于是要求 'a 至少跟 s 的生命期一样长。此刻 Rust 又遇到了跟之前一样矛盾的限制：'a 不能比 x“长寿”，但至少要跟 s 一样“长寿”。不存在这样的生命期，所以 Rust 会拒绝编译。又避免了一场灾难！

把带有生命期参数的类型放到其他类型中又会怎样？

```
struct T {
    s: S // 这样不行
}
```

Rust 仍然会质疑，就像试图不给 S 指定生命期却要让它包含引用一样：

```
error[E0106]: missing lifetime specifier
 --> references_in_nested_struct.rs:8:8
  |
8 |     s: S // 这样不行
  |        ^ expected lifetime parameter
```

这里也不能漏掉 S 的生命期参数：Rust 需要知道 T 的生命期与它包含的 S 中的引用的生命期是什么关系，以便对 T 应用跟对 S 和纯引用相同的检查。

可以给 s 声明 'static 生命期：

```
struct T {
    s: S<'static>
}
```

这样一来，s 字段就只能借用那些在整个程序执行期间都会存活的值。这样有点受限，不过确实意味着 T 不可能借用局部变量了；除此之外，对 T 的生命期没有什么特殊的约束。

此外，也可以给 T 一个自己的生命期参数，同时传给 S：

```
struct T<'a> {
    s: S<'a>
}
```

通过声明生命期参数 'a 并在 s 的类型中也使用该生命期，Rust 就可以将 T 值的生命期与其 S 中所保存引用的生命期关联起来。

前面展示过通过函数签名可以知道给它传什么引用。现在又在类型上看到了类似的情形：通过类型的生命期参数，可以知道该类型是否包含具有相应（非 'static）生命期的引用，以及那些引用的情况如何。

比如，假设有一个解析函数，其参数是一个字节切片，它会返回保存解析结果的一个结构：

```
fn parse_record<'i>(input: &'i [u8]) -> Record<'i> { ... }
```

根本不用去看 Record 类型的定义，就可以知道如果从 parse_record 接收到 Record，那么其中包含的引用一定指向我们传入的输入缓冲区，而不会是别处（'static 值除外）。

事实上，Rust 之所以要求包含引用的类型不能省略生命期参数，正是为了将这种内部行为外化表示出来。Rust 并非不可以给结构体中的每个引用构造生命期，从而帮你省掉把它们写出来的麻烦。Rust 早期的版本就是这么做的，但开发者反映这样比较乱：知道一个值从另一个值借用了什么还是很有用的，特别是在排错的时候。

不仅仅引用和像 S 这样的类型有生命期，Rust 中的所有类型都有生命期，包括 i32 和 String。大多数是简单的 'static，也就是说，你想让它们的生命期有多长就有多长。比如，一个 Vec<i32> 值，它是独立的，不需要在任何特定变量超出作用域时被清除。但是像 Vec<&'a i32> 这样生命期必须包含在 'a 中的类型，则必须在其引用的值仍然有效时先被清除。

5.2.6 不同的生命期参数

假设我们定义了一个结构体，其中包含两个引用：

```
struct S<'a> {
    x: &'a i32,
    y: &'a i32
}
```

这两个引用使用相同的生命期 'a。如果你的代码想像下面这样做则可能会有问题：

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
```

以上代码不会创建悬空指针。y 的引用保存在 s 中，而 s 会先于 y 被销毁。x 的引用最终保存在 r 中，而 r 也不会比 x“活”得更久。

可是，如果你试图用较早版本的编译器编译这段代码，Rust 就会抱怨说 y“活”得不够长，尽管明显够长。为什么 Rust 会担心呢？仔细分析一下代码，你会发现它这样做的道理。

- S 的两个字段都是引用，且拥有相同生命期 'a，因此 Rust 必须找到一个同时对 s.x 和 s.y 都合适的生命期。
- 赋值 r = s.x 要求 'a 涵盖 r 的生命期。

- 以 &y 初始化 s.y，要求 'a 的生命期不能长过 y。

没有比 y 的作用域短却比 r 的作用域长的生命期，这两个约束不可能同时满足。于是 Rust 停了下来。

出现这个问题，原因在于 S 的两个引用有相同的生命期 'a。修改 S 的定义，让两个引用分别拥有不同的生命期，一切就迎刃而解了：

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}
```

如此一改，s.x 和 s.y 就有了独立的生命期。对 s.x 的操作不会影响到 s.y，因此前面的约束就很容易满足了：'a 可以等于 r 的生命期，而 'b 可以等于 s 的生命期（'b 也可以等于 y 的生命期，不过 Rust 倾向于选择可用的最短生命期）。最终一切都顺理成章。

函数签名也有类似的作用。假设有如下函数：

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r } // 可能太严苛了
```

这里，两个引用参数的生命期都是 'a，没必要像前面一样限制调用。如果这样确实有问题，那可以让参数拥有不同的生命期：

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r }  // 宽松多了
```

放宽限制的缺点在于，过多的生命期参数会导致类型和函数签名复杂难读。作为本书作者，我通常会先尝试最简单的定义，如果编译无法通过就放松限制，直到编译通过。因为 Rust 不会放过不安全的代码，所以等着它告诉你哪里有问题你再修改是一个完全可以接受的策略。

5.2.7 省略生命期参数

到目前为止，本书展示的很多函数以引用为返回值或参数，但通常不需要写出生命期参数。无论写还是不写，生命期一直都在。Rust 会在生命期参数明显合理时让你省略它们。

在最简单的情况下，如果函数不返回任何引用（或者其他需要生命期参数的类型），那永远也不需要写出参数的生命期。Rust 会在每个需要的地方加上明确的生命期。比如：

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}

fn sum_r_xy(r: &i32, s: S) -> i32 {
    r + s.x + s.y
}
```

上面函数的签名完整地写出来应该是这样的：

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

如果函数确实要返回引用或其他带生命期参数的类型，那 Rust 也会尽量让消除歧义的过程简化。如果函数参数中只出现了一个生命期，那 Rust 会假定返回值中的生命期都是该生命期：

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
    (&point[0], &point[2])
}
```

这个函数包含所有生命期的完整签名如下：

```
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

如果函数参数中有多个生命期，那么自然没理由认为哪个生命期更适合返回值，此时 Rust 会要求你自己写清楚。

最后，还有一种可能的简写形式，那就是如果函数是某个类型的方法，而它又接收引用形式的 `self` 参数，那么平衡就被打破了：Rust 会假定 `self` 的生命期就是返回值需要的生命期。（`self` 参数引用的是调用当前方法的值，相当于 C++、Java 或 JavaScript 中的 `this`，或者 Python 中的 `self`。关于方法，9.5 节会讨论。）

比如，可以这样写：

```
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
        for i in 0 .. self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}
```

这里 `find_by_prefix` 方法的完整签名如下：

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) -> Option<&'a String>
```

Rust 假定无论你借用的是什么，都是从 `self` 中借用的。

说到底，这些简写形式就是为了少吓人。如果简写不符合你的预期，那完全可以自己明确写出生命期。

5.3 共享与修改

本书此前讨论的都是 Rust 如何确保引用永远不会指向已经离开作用域的变量。但是还有别的方式可能引入悬空指针。比如这样：

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v;  // 将向量转移到aside
r[0];           // 不行：使用v，可是它已经变成未初始化了
```

给 `aside` 赋值会转移向量，导致 `v` 变成未初始化，而 `r` 也变成了悬空指针，如图 5-7 所示。

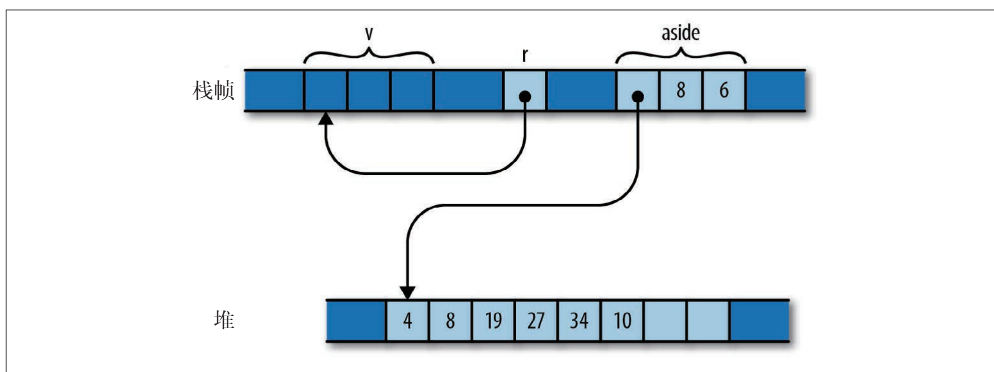


图 5-7：r 引用的向量已经转移了

虽然 v 在 r 的整个生命期内都存在，但问题是 v 的值已经转移到别处，v 变成未初始化了，而 r 还引用着它。自然，Rust 也捕获了这个错误：

```
error[E0505]: cannot move out of `v` because it is borrowed
  --> references_sharing_vs_mutation_1.rs:10:9
   |
9  |     let r = &v;
   |             - borrow of `v` occurs here
10 |     let aside = v;  // 将向量移到一边
   |         ^^^^^ move out of `v` occurs here
```

终其整个生命期，共享引用的目标值都是只读的：不能重新给它赋值或转移该值。而在上面的代码中，r 的生命期内发生了转移向量的操作，Rust 当然要拒绝。把代码改成下面这样就没事了：

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];       //  可以：向量没动
}
let aside = v;
```

这一次，r 先离开作用域，引用的生命期在 v 转移之前结束，一切相安无事。

下面来看另一种破坏方式。假设有一个函数，它拿一个切片的元素来扩展一个向量：

```
fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

这是标准库向量方法 `extend_from_slice` 的缩水版（少了很多优化）。通过它可以基于别的向量或数组的切片构建一个向量：

```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];
```

```
extend(&mut wave, &head);   // 用另一个向量扩展wave
extend(&mut wave, &tail);   // 用数组扩展wave

assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);
```

这样就构造了正弦波的一个周期。如果想再增加一个波形，能不能追加向量本身呢？

```
extend(&mut wave, &wave);
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,
                      0.0, 1.0, 0.0, -1.0]);
```

乍一看这样没问题。但别忘了，在给向量追加元素时，如果其缓冲区满了，就必须分配一块更大的空间。假设 wave 开始时的空间能容纳 4 个元素，因此在 extend 想添加第五个时必须分配更大的缓冲区。此时的内存如图 5-8 所示。

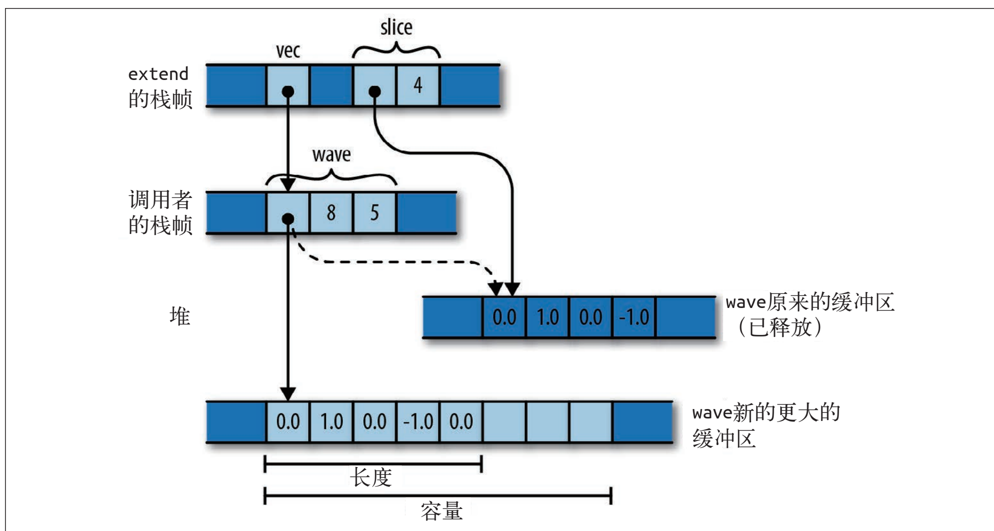


图 5-8：由于向量重分配空间导致切片变成悬空指针

这个 extend 函数的 vec 参数借用了（调用者拥有的）wave，而 wave 给自己重新分配了一个能容纳 8 个元素的新缓冲区。但 slice 参数仍然指向原来 4 个元素的缓冲区，该内存已被清除。

这种问题并非 Rust 独有：在很多语言中，修改集合的同时还指向集合很容易出问题。在 C++ 中，std::vector 规范也有警告"[向量缓冲区的] 重新分配会导致所有指向该序列中元素的引用、指针和迭代器失效"。类似地，针对修改 java.util.Hashtable 对象，Java 也在文档中指出：

> 在创建迭代器之后，除非通过迭代器自身的 remove 方法对 Hashtable 进行结构性修改，否则迭代器会抛出 ConcurrentModificationException。

查找这种 bug 之所以特别难，主要是因为其并不经常发生。测试的时候，向量可能始终都有充足的空间，缓冲区不会重新分配，因而问题可能永远不会暴露出来。

不过，Rust 会在编译时报告 extend 调用有问题：

```
error[E0502]: cannot borrow `wave` as immutable because it is also borrowed as mutable
 --> references_sharing_vs_mutation_2.rs:9:24
  |
9 |     extend(&mut wave, &wave);
  |            ----    ^^^^- mutable borrow ends here
  |            |       |
  |            |       immutable borrow occurs here
  |            mutable borrow occurs here
```

Rust 的意思是，可以借用向量的可修改引用，也可以借用对其元素的共享引用，但这两个引用的生命期不能重叠。在我们的代码中，这两个引用的生命期都包含 extend 调用，因此 Rust 拒绝编译。

归根结底，以上错误都源于违背了 Rust 关于修改和共享的规则。

- **共享访问是只读访问**。共享引用借用的值是只读的。在共享引用的整个生命期内，**任何事物**都不能修改其引用目标，也不能修改其引用目标可触及的值。结构中不存在指向任何目标的可修改引用，所有者是只读的……实际上这个值冻结了。
- **可修改访问是排他访问**。可修改引用借用的值只能通过该引用访问。在可修改引用的整个生命期内，没有其他路径可触及其引用目标，或触及其引用目标可触及的值。唯一能够与可修改引用的生命期重叠的，就是从该可修改引用自身借用的引用。

Rust 报告 extend 例子违反了第二条规则：因为已经向 wave 借用了一个可修改引用，所以该引用必须是可触及这个向量及其元素的唯一途径。而对切片的共享引用本身是可抵达向量元素的另一条路径，这违反了第二条规则。

不过，Rust 也可以认为我们的 bug 违反了第一条规则：因为已经借用了对 wave 元素的一个共享引用，所以无论元素还是 vec 本身都是只读的。此时就不能再向一个只读的值借用可修改引用了。

这两种引用以不同方式影响我们沿所有权路径操作目标值，以及操作通过目标值可达的其他值，如图 5-9 所示。

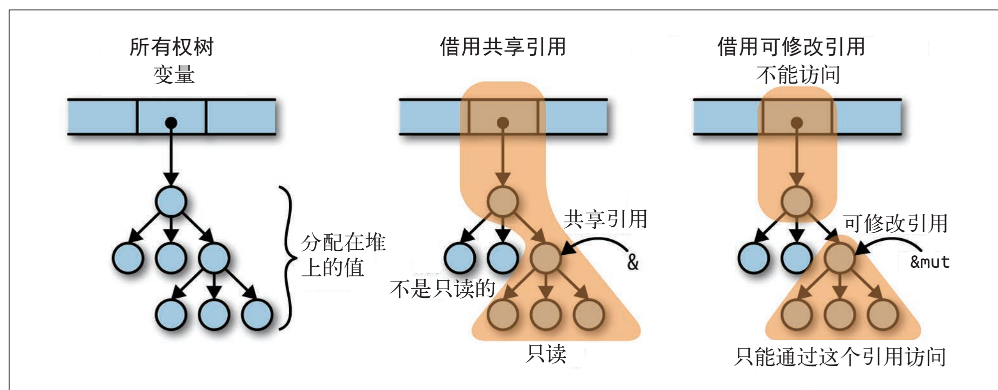


图 5-9：借用引用会影响我们操作同一个所有权树中的其他值

注意在这两种情况下，抵达引用目标的所有权路径都不会因引用的生命期而改变。对于共享借用，这条路径是只读的；对于可修改借用，这条路径完全不通。因此程序没有任何途径做使引用无效的事。

这些原理可以通过下面的简单的例子来说明：

```
let mut x = 10;
let r1 = &x;
let r2 = &x;       // 可以：允许多次共享借用
x += 10;           // 错误：不能给x赋值，因为它已经被借用了
let m = &mut x;    // 错误：不能借用x的可修改引用，因为它已经
                   // 借出了不可修改引用

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y; // 错误：不能借两次可修改引用
let z = y;       // 错误：不能用y，因为它已经借出了可修改引用
```

从共享引用借用共享引用是可以的：

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;        // 可以：共享引用可以再借用为共享引用
let m1 = &mut r.1;    // 错误：共享引用不能再借用为可修改引用
```

从可修改引用再借用可修改引用也是可以的：

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // 可以：可修改引用可以再借用为可修改引用
*m0 = 137;
let r1 = &m.1;          // 可以：可修改引用可以再借用为共享引用，
                        // 且不与（可修改引用）m0重叠
v.1;                    // 错误：禁止通过其他路径访问
```

这些限制相当严格。回头再看看 `extend(&mut wave, &wave)` 调用，并没有简易可行的办法修改代码以实现我们的意图。而且 Rust 会在任何地方应用这些规则，比如我们借用了 `HashMap` 中一个键的共享引用，直到共享引用的生命期结束，才能再借用对这个 `HashMap` 的可修改引用。

不过有一个很正当的理由：设计出一种集合，让它支持无限制、同时迭代和修改还是很困难的，经常会因此排除更简单、有效的实现。Java 的 `Hashtable` 和 C++ 的 `vector` 没有自找麻烦，当然 Python 的字典、JavaScript 的对象也没有对这种存取行为给出明确定义。JavaScript 对其他集合类型有明确定义，但实现会比较烦琐。C++ 的 `std::map` 承诺插入新元素不会影响指向映射中其他元素的指针，但因为有了这个承诺，其标准库排除了类似 Rust 的 `BTreeMap` 这样缓存效率更高的设计（Rust 的 `BTreeMap` 在树的每个节点都保存多个元素）。

下面再看看上述规则可以捕获的另一种 bug 的例子。下面的 C++ 代码用于管理一个文件描述符。为简单起见，这里只给出一个构造器和一个复制赋值操作符，省略错误处理：

```
struct File {
  int descriptor;

  File(int d) : descriptor(d) { }
```

```
  File& operator=(const File &rhs) {
    close(descriptor);
    descriptor = dup(rhs.descriptor);
    return *this;
  }
};
```

赋值操作符相当简单，但在下面这种情况下会失败：

```
File f(open("foo.txt", ...));
...
f = f;
```

如果把 `File` 赋值给它自己，`rhs` 和 `*this` 就是同一个对象，`operator=` 恰好会在给 `dup` 传递描述符之前关闭它。换句话说，我们销毁了本来想复制的资源。

在 Rust 中，实现类似功能的代码如下：

```
struct File {
    descriptor: i32
}

fn new_file(d: i32) -> File {
    File { descriptor: d }
}

fn clone_from(this: &mut File, rhs: &File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}
```

（这不是 Rust 的习惯写法。第 9 章会讨论为 Rust 类型提供自己的构造器函数和方法的更好方法。前面的代码只是很适合作为示例而已。）

如果再把使用 `File` 的 Rust 代码写出来，应该是：

```
let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);
```

当然，Rust 会直接拒绝编译它：

```
error[E0502]: cannot borrow `f` as immutable because it is also
borrowed as mutable
  --> references_self_assignment.rs:18:25
   |
18 |     clone_from(&mut f, &f);
   |                     -   ^- mutable borrow ends here
   |                     |   |
   |                     |   immutable borrow occurs here
   |                     mutable borrow occurs here
```

这个错误似曾相识。原来两个经典的 C++ bug（无法处理自赋值和使用失效的迭代器）背后是同一类 bug！这两种情况下，代码都会假定自己在修改一个值的时候参照的是另一个值，而实际上它们是同一个值。如果你在使用 C/C++ 的 `memcpy` 或 `strcpy` 时曾意外让它们的源和目标重叠，那你就碰到了这种 bug 的第三种表现形式。Rust 通过将可修改存取限制为排他操作，帮我们避免了一大类日常错误。

在编写并发代码时，共享和可修改引用不相混合的价值会体现得淋漓尽致。只有某个值在线程间既可修改又共享时，数据争用才会发生，而这恰恰是 Rust 的引用规则所致力消除的。不涉及 `unsafe` 代码的并发 Rust 程序，**从根基上**就不会存在数据争用。第 19 章在讨论并发编程时会深入讨论这一点。不过简单概括一下就是，用 Rust 编写并发程序比用大多数其他语言更容易。

Rust 的共享引用与 C 的常量指针

Rust 共享引用给人的第一印象是它非常像 C 和 C++ 的常量指针。然而，Rust 共享引用的规则要严格得多。比如，对于如下 C 代码：

```
int x = 42;             // 整数变量，不是常量
const int *p = &x;      // 指向整数常量的指针（常量指针）
assert(*p == 42);
x++;                    // 直接修改变量
assert(*p == 43);       // “不变”目标的值也变了
```

`p` 是一个 `const int *` 意味着不能通过 `p` 来修改它引用的值：`(*p)++` 是不行的。不过，通过 `x` 也可以直接访问到同一个值，`x` 又不是常量，直接修改它的值就好了。C 家族的 `const` 关键字有其用途，但常量不尽然。

在 Rust 中，共享引用禁止对引用的值进行任何修改，直到其生命期结束：

```
let mut x = 42;         // i32变量，不是常量
let p = &x;             // 对i32值的共享引用
assert_eq!(*p, 42);
x += 1;                 // 错误：因为被借用了，所以不能给x赋值
assert_eq!(*p, 42);     // 去掉赋值，这个断言为true
```

为保证值不变，需要跟踪抵达该值的所有路径，确保要么不允许修改，要么根本就到不了该值。C 和 C++ 编译器在检查这方面时对指针过于宽松。Rust 的引用则始终与特定生命期贴合，因而编译时检查是可行的。

5.4 征服对象之海

自 20 世纪 90 年代自动内存管理兴起之后，所有程序默认的架构就是**对象之海**，如图 5-10 所示。

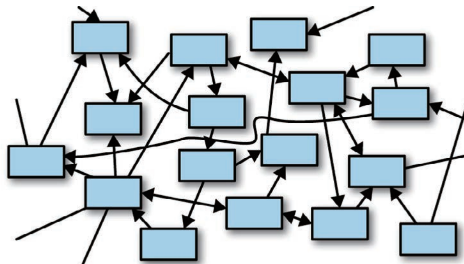


图 5-10：对象之海

如果语言有垃圾收集，而且没经过设计就开始编写程序，就是这个样子。一直以来，我们构建的系统都是如此。

这样的架构有很多图中没有显示的优点。比如，可以快速启动一个项目，方便验证各种想法，等等。但随着时间推移，几年之后，你会毫不费力地发现这个项目必须完全重写了（暗合了 AC/DC 的“通往地狱的快速路”）。

当然，它也有缺点。像这样对象之间可以无限地相互依赖，程序会难于测试、演进，更不要说组件隔离了。

Rust 特别迷人的一个地方就在于，其所有权模型给这条“通往地狱的快速路”设置了减速带。在 Rust 中构造一个循环引用（两个值互相包含对对方的引用）颇费周折。你得使用一种智能指针，比如 `Rc`，还要用到内部修改能力（目前还没讲到这一块）。Rust 更喜欢指针、所有权和数据流单向通过整个系统，如图 5-11 所示。

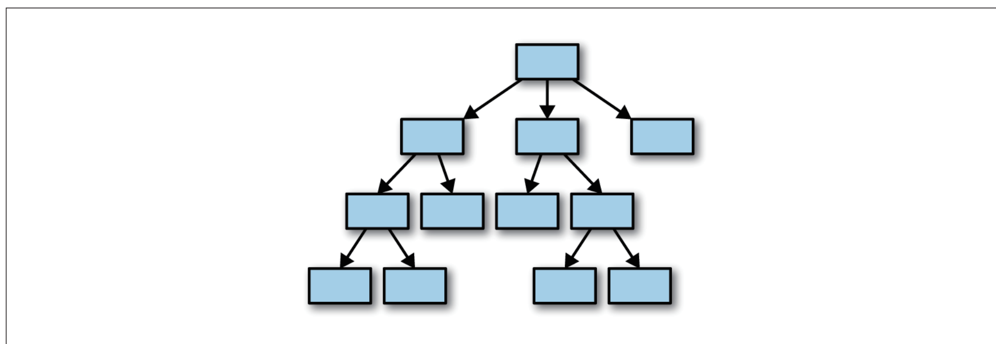


图 5-11：值的树

之所以在这个当口指出这一点，是因为看完这一章之后，大家很容易想打破约束去创造一个“结构体之海”，用 `Rc` 智能指针把它们连接起来，最终复现自己熟悉的各种面向对象反模式。你不会马上得逞，Rust 的所有权模型总会给你制造一些麻烦。解决之道是做一些事前设计，构建更好的程序。

Rust 所做的一切，都是为了把理解程序的痛苦从未来拉回到现在。不可思议的是，其效果很好：Rust 不仅可以强迫你理解为什么自己的程序是线程安全的，甚至还会要求你在更高的层面上进行一些架构设计。

第 6 章

表达式

Lisp 程序员知道所有东西的值，却不知道那些东西的计算成本。

——Alan Perlis，警句 #55

本章将介绍 Rust 的**表达式**，它也是 Rust 函数体的主角。有些概念，比如闭包、迭代器，都太深奥了，稍后本书会分别用一章去讨论。眼下，我们尽量用较少的篇幅来多讲一些语法。

6.1　表达式语言

Rust 虽然看起来和 C 家族的语言很像，但这只是它的一个策略。在 C 中，**表达式**和**语句**有着鲜明的区别，这是表达式：

```
5 * (fahr-32) / 9
```

而这是语句：

```
for (; begin != end; ++begin) {
    if (*begin == target)
        break;
}
```

表达式有值，语句却没有。

Rust 是所谓的**表达式语言**。这意味着它遵循一种比较早期的传统，可以追溯到 Lisp，当时表达式包打天下。

在 C 中，`if` 和 `switch` 是语句。它们不产生值，也不能用在表达式中间。而在 Rust 中，`if` 和 `match` **可以**产生值。第 2 章已经展示了产生数值的 `match` 表达式：

```
pixels[r * bounds.0 + c] =
    match escapes(Complex { re: point.0, im: point.1 }, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
```

Rust 的 if 表达式可以用来初始化变量：

```
let status =
    if cpu.temperature <= MAX_TEMP {
        HttpStatus::Ok
    } else {
        HttpStatus::ServerError // 服务器错误
    };
```

Rust 的 match 表达式可以作为参数传给函数或宏：

```
println!("Inside the vat, you see {}.",
    match vat.contents {
        Some(brain) => brain.desc(),
        None => "nothing of interest"
    });
```

这也解释了为什么 Rust 没有 C 的三元操作符（expr1 ? expr2 : expr3）。在 C 中，三元操作符是对 if 语句在表达式层面的一个快捷等价物。但在 Rust 中，它反而显得多余，因为 if 表达式两种情形都可以处理。

C 中的大多数控制流工具是语句。而在 Rust 中，它们全是表达式。

6.2 块与分号

代码块，同样也是表达式。块产生值，其可以用于任何需要值的地方：

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

Some(author) => 之后的代码是一个简单的表达式 author.name()。而 None => 之后的代码是一个块表达式。在 Rust 中它们没有区别。这里块的值就是最后一个表达式 ip.to_string() 的值。

注意，那个表达式末尾没有分号。大多数 Rust 代码末尾不是分号就是花括号，就跟 C 或 Java 一样。而如果一个块看起来像 C 代码，分号出现在了熟悉的位置，那么它也会像 C 的块一样运行，其值将为 ()。正如第 2 章提到的，在块的最后一行省略分号，就会让这个块产生一个值，即最后一个表达式的值。

在某些语言，特别是 JavaScript 中，分号并不是强制要写的，不写的话语言会帮你填上，

这是 JavaScript 中的一点小便利。但这里不一样。在 Rust 中，分号是有意义的。

```
let msg = {
    // let声明：分号是必需的
    let dandelion_control = puffball.open();

    // 表达式+分号：方法调用，返回值被清除
    dandelion_control.release_all_seeds(launch_codes);

    // 表达式不带分号：方法被调用，返回值保存于msg中
    dandelion_control.get_status()
};
```

块的一个精妙特性是既可以包含声明又可以在末尾产生值。任何人都可以很快习以为常。唯一的缺点是：在意外漏掉分号时，会导致一个奇怪的错误消息。

```
...
if preferences.changed() {
    page.compute_size()   // 噢，漏掉了分号
}
...
```

如果是在 C 或 Java 程序中犯这个错，编译器会直接告诉你漏掉了分号。而 Rust 会说：

```
error[E0308]: mismatched types
  --> expressions_missing_semicolon.rs:19:9
   |
19 |         page.compute_size()   // 噢，漏掉了分号
   |         ^^^^^^^^^^^^^^^^^^^ expected (), found tuple
   |
   = note: expected type `()`
              found type `(u32, u32)`
```

Rust 假设你是有意省略分号的，它不会考虑这可能只是个打字错误。结果是一个让人糊涂的错误消息。以后只要看到 `expected type`()``，记着先看看是不是漏写了分号。

空语句可以出现在块中。空语句就是一个孤立的分号，只有它自己：

```
loop {
    work();
    play();
    ;            // <-- 空语句
}
```

Rust 遵循 C 的传统，允许出现这种情况。空语句除了传达一丝淡淡的惆怅外什么也不干。这里提到它也只是出于圆满的考虑。

6.3 声明

除了表达式和分号，块还可以包含多个任意声明。最常见的是用于声明局部变量的 `let` 声明：

```
let name: type = expr;
```

其中，类型和初始值是可选的，分号是必需的。

let 声明可以只声明变量而不初始化它。然后再通过赋值来初始化变量。这样偶尔有用，因为有时候需要在某种控制流结构中间初始化变量：

```
let name;
if user.has_nickname() {
    name = user.nickname();
} else {
    name = generate_unique_name();
    user.register(&name);
}
```

这里有两种不同的初始化局部变量 name 的方式，但不管是哪种方式都只能初始化一次，因此 name 不需要声明为 mut。

在初始化之前使用变量是错误的（这跟值被转移之后还使用它是类似的，Rust 希望我们只在它们存在时使用值）。

偶尔，你也会看到像是重新声明一个已有变量的代码，比如：

```
for line in file.lines() {
    let line = line?;
    ...
}
```

以上代码等价于：

```
for line_result in file.lines() {
    let line = line_result?;
    ...
}
```

这个 let 声明创建了一个新的、不同的变量，类型也不一样。line_result 的类型是 Result<String, io::Error>。而第二个变量 line 是一个 String。第二个变量跟第一个变量使用相同的名字是合法的。本书在这种情况下只使用 _result 后缀，因此所有变量都有不同的名字。

块里也可以包含**特性项声明**。所谓特性项（item）指的是任何可以在程序或模块的全局中出现的声明，比如 fn、struct 或 use。

后面几章会详细讨论特性项。现在，fn 已经是个很不错的例子了。任何块中都可以包含 fn：

```
use std::io;
use std::cmp::Ordering;

fn show_files() -> io::Result<()> {
    let mut v = vec![];
    ...

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering {
        a.timestamp.cmp(&b.timestamp)    // 首先，比较时间戳
            .reverse()                   // 最新的文件排在前面
            .then(a.path.cmp(&b.path))   // 比较路径排除
    }
```

```
    v.sort_by(cmp_by_timestamp_then_name);
    ...
}
```

在块中声明的 fn，其作用域是整个块。换句话说，它可以在整个块内**被使用**。但嵌套的 fn 不能访问该作用域中的局部变量或参数。比如，函数 cmp_by_timestamp_then_name 不能直接使用 v。（Rust 也有闭包，闭包是可以访问封闭它的作用域的。详见第 14 章。）

块中甚至可以包含一个完整的模块。这似乎有点过了：真的需要把语言的**每一**块都嵌套在其他块中吗？不过程序员（特别是使用宏的程序员）总能把语言提供的每个特性都派上用场。

6.4 if与match

if 表达式的形式我们很熟悉：

```
if condition1 {
    block1
} else if condition2 {
    block2
} else {
    block_n
}
```

每个 condition 都必须是一个 bool 型的表达式。而且一如既往，Rust 不会隐式地把数值或指针转换为布尔值。

与 C 不同，围绕条件的圆括号不是必需的。事实上，rustc 在发现不必要的圆括号时会给出警告。但花括号是必需的。

else if 块和最后的 else 块都是可选的。一个没有 else 块的 if 表达式，就像它有一个空的 else 块一样。

match 表达式有点类似 C 的 switch 语句，但更灵活。来看一个简单的例子：

```
match code {
    0 => println!("OK"),
    1 => println!("Wires Tangled"),
    2 => println!("User Asleep"),
    _ => println!("Unrecognized Error {}", code)
}
```

这也是 switch 语句可以胜任的。实际上，根据 code 的值，4 个 match 表达式的分支中只有 1 个会执行。通配模式 _ 匹配任何值，对应 default: 条件。

编译器可以使用跳转表（jump table）来优化这种 match 表达式，就像 C++ 中的 switch 语句一样。如果 match 的每个分支都产生一个常量值，那么也可以应用同样的优化。此时，编辑器会构建一个这些值的数组，而 match 会被编译为对数组的访问。除了边界检查，编译后的代码中根本没有分支。

功能丰富的 match 源自对各种各样**模式**的支持，这些模式可以用在每个分支 => 的左侧。上面，每个模式就是一个常量整数。我们也看到了能区分两种 Option 值的 match 表达式：

```
match params.get("name") {
    Some(name) => println!("Hello, {}!", name),
    None => println!("Greetings, stranger.")
}
```

对于模式所能做的事，这只能算是管中窥豹。模式可以匹配的值多种多样：可以解包（unpack）元组，可以匹配结构体的个别字段，可以追索引用，可以借用一个值的某一部分，等等。Rust 的模式本身就是一门小型语言。第 10 章会专门用一定篇幅来讨论模式。

`match` 表达式的通用形式如下：

```
match value {
    pattern => expr,
    ...
}
```

如果 `expr` 是一个块，则后面的逗号可以去掉。

Rust 会依次用每个模式去匹配 `value`，从第一个模式开始。如果哪个模式匹配，就对相应的 `expr` 求值，而这个 `match` 表达式结束；后面的模式就不再继续检查了。所有模式中必须至少有一个匹配。Rust 会拒绝不能涵盖所有可能值的 `match` 表达式：

```
let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11
};  // 错误：模式不够全面
```

`if` 表达式的所有块都必须产生相同类型的值：

```
let suggested_pet =
    if with_wings { Pet::Buzzard } else { Pet::Hyena }; // 可以

let favorite_number =
    if user.is_hobbit() { "eleventy-one" } else { 9 }; // 错误

let best_sports_team =
    if is_hockey_season() { "Predators" }; // 错误
```

（最后一个例子错误，是因为 7 月份不是曲棍球比赛季，结果将为 `()`。）

类似地，`match` 表达式的所有分支也都必须返回相同类型的值：

```
let suggested_pet =
    match favorites.element {
        Fire => Pet::RedPanda,
        Air => Pet::Buffalo,
        Water => Pet::Orca,
        _ => None // 错误：不兼容的类型
    };
```

if let

`if` 还有一种形式，就是 `if let` 表达式：

```
if let pattern = expr {
    block1
} else {
    block2
}
```

这里的 expr 要么匹配 pattern，运行 block1，要么不匹配 pattern，运行 block2。有时候，这是从 Option 或 Result 中取得数据的一种便捷方式：

```
if let Some(cookie) = request.session_cookie {
    return restore_session(cookie);
}

if let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    politely_accuse_user_of_being_a_robot();
} else {
    session.mark_as_human();
}
```

并不是说这里**必须**使用 if let，因为 match 也完全可以代替 if let。if let 表达式只是对只有一个模式的 match 的简写：

```
match expr {
    pattern => { block1 }
    _ => { block2 }
}
```

6.5 循环

有 4 种循环表达式：

```
while condition {
    block
}

while let pattern = expr {
    block
}

loop {
    block
}

for pattern in collection {
    block
}
```

循环在 Rust 中是表达式，但它们不会产生有用的值。循环的值是 ()。

while 循环的行为与 C 类似，除了 condition 必须是 bool 类型。

while let 循环跟 if let 类似。在每次循环迭代开始时，expr 的值要么匹配给定的

pattern，运行后面的块，要么不匹配给定的 pattern，退出循环。

loop 用于编写无穷循环。它后面的 block 会永远重复执行（或直至遇到一个 break 或 return，或者线程诧异）。

for 循环对 collection 表达式求值，然后该集合中的每个值分别对 block 求值。支持的集合类型有很多。以下是标准的 C 循环：

```
for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}
```

Rust 中则是这样写的：

```
for i in 0..20 {
    println!("{}", i);
}
```

跟 C 中一样，最后一个打印的数值是 19。

.. 操作符产生一个**范围**（range），即一个拥有两个字段（start 和 end）的简单结构体。0..20 跟 std::ops::Range { start: 0, end: 20 } 是一样的。Range 可用于循环是因为它是一个可迭代类型，其实现了第 15 章讨论的 std::iter::IntoIterator 特型。标准的集合，比如数组和切片，都是可迭代的。

与 Rust 的转移语义一致，for 循环每迭代一个值就会用掉一个值：

```
let strings: Vec<String> = error_messages();
for s in strings {                          // 每个String都在这里转移到s……
    println!("{}", s);
}                                           // ……并在这里被清除
println!("{} error(s)", strings.len());    // 错误：使用被转移的值
```

这可能会带来不便。简单的补救方式是迭代对集合的引用。这样，循环变量就是对集合中每一项的引用：

```
for rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}
```

这里 &strings 的类型是 &Vec<String>，rs 的类型是 &String。

迭代 mut 引用，则循环变量拿到的也是 mut 引用：

```
for rs in &mut strings { // rs的类型是&mut String
    rs.push('\n'); // 给每个字符串添加一个换行符
}
```

第 15 章还会更详尽地介绍 for 循环，并展示使用迭代器的很多其他方式。

break 表达式用于退出闭合循环。（在 Rust 中，break 只能用在循环中。因为不像 switch 语句，match 表达式用不着它。）

continue 表达式跳到循环的下一次迭代：

```
// 读取数据，每次一行
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
       // 跳到循环顶部，取得输入的下一行
       continue;
    }
    ...
}
```

在 for 循环中，continue 会前进到集合中的下一个值。如果已经没有值了，则退出循环。类似地，在 while 循环中，continue 会再次检查循环条件，如果为假，则退出循环。

循环可以加上生命期**标签**。在下面的例子中，'search: 是外部 for 循环的标签。因此 break 'search 要退出的不是内部循环，而是外部循环。

```
'search:
for room in apartment {
    for spot in room.hiding_spots() {
        if spot.contains(keys) {
            println!("Your keys are {} in the {}.", spot, room);
            break 'search;
        }
    }
}
```

标签也可以跟 continue 一起用。

6.6 return表达式

return 表达式退出当前函数，向调用者返回一个值。

无值 return 表达式是 return () 的简写：

```
fn f() {       // 省略返回类型：默认为()
    return;    // 省略返回值：默认为()
}
```

与 break 表达式类似，return 可以放弃正在进行的工作。比如，第 2 章曾使用 ? 操作符检查可能失败的函数调用：

```
let output = File::create(filename)?;
```

并解释说这是对一个 match 表达式的简写：

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err)
};
```

这里代码先调用 File::create(filename)，如果返回 Ok(f)，那么整个 match 表达式求值为 f，因此 f 会保存在 output 中。然后继续执行 match 表达式后面的下一行代码。

否则，就会匹配 `Err(err)` 并命中 `return` 表达式。这时候，对 `match` 表达式求值以确定变量 `output` 的值已经不重要了，因为函数调用出错会终止后面的所有工作，并退出闭合函数，返回从 `File::create()` 调用获得的错误。

7.2.4 节在介绍传播错误时，会更全面地介绍 ? 操作符。

6.7 为什么Rust有循环

Rust 编译器有几个部分会通过你的程序分析控制流。

- Rust 检查贯穿函数的每条路径，确保返回值为正确类型。为了正确地完成这个检查，它需要知道是否可能到达函数末尾。
- Rust 检查局部变量永远不会在未始终化时被使用。因此必须检查贯穿函数的每一条路径，以确保不会抵达一个变量尚未经过初始化就被使用的地方。
- Rust 会对无法抵达的代码给出警告。如果贯穿函数的路径**没有**一条能抵达，那么相应的代码就是无法抵达的。

以上这些称为**流敏感**（flow-sensitive）分析。这没什么新鲜的，Java 很多年前就有“明确赋值”（definite assigment）分析了，Rust 的也类似。

在强制执行这些规则时，语言必须在简单（simplicity）和机巧（cleverness）之间取得平衡。前者可以让程序员有时候更容易明白编译器在说什么，后者有助于减少误报及避免拒绝实际上非常安全的程序。Rust 追求简单，其流敏感分析压根不会检查循环条件，而只是假设程序中的任何条件不是 `true` 就是 `false`。

于是，Rust 可能会拒绝某些安全的程序：

```
fn wait_for_process(process: &mut Process) -> i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
} // 错误：并非所有控制路径都返回值
```

这个错误是假的。实际上，没有返回值不可能抵达函数的末尾。

`loop` 表达式就是作为解决这个问题的“心口如一”的方案给出的。

Rust 的类型系统也受控制流的影响。前面说过，`if` 表达式的所有分支必须是相同的类型。如果把这个规则强加给以 `break` 或 `return` 表达式结尾的块、无穷 `loop`、对 `panic!()` 或 `std::process::exit()` 的调用，则是不明智的。这些表达式共有的特点是它们都不以惯常的方式结束，不返回值。`break` 或 `return` 会突然退出当前块，无穷循环永远也不会结束……

因此在 Rust 中，这些表达式没有常规的类型。不正常结束的表达式通常被指定为特殊类型 `!`，它们不受其他类型需要遵从的规则的约束。在 `std::process::exit()` 的函数签名中可以看到 `!`：

```
fn exit(code: i32) -> !
```

! 的意思是 exit() 永远也不会返回，它是一个**发散函数**（divergent function）。

使用同样的语法也可以自定义发散函数，在某些情况下这是自然而然的事：

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) -> ! {
    socket.listen();
    loop {
        let s = socket.accept();
        handler.handle(s);
    }
}
```

当然，如果这个函数可以正常返回，那么 Rust 会认为它是一个错误。

本章关于控制流的部分至此就告一段落了。接下来介绍 Rust 函数、方法和操作符。

6.8　函数与方法调用

跟在许多其他语言中一样，调用函数与方法的语法在 Rust 中是一样的：

```
let x = gcd(1302, 462); // 函数调用

let room = player.location(); // 方法调用
```

这里第二个例子中，player 是编造的类型 Player 的变量，这个类型有一个编造的 .location() 方法。（第 9 章在讨论用户定义类型时会介绍如何定义自己的方法。）

Rust 通常会将引用与它们指向的值截然分开。如果你给期望 i32 的函数传入一个 &i32，那就是个类型错误。但 . 操作符会让这个规则放宽松一些。在方法调用 player.location() 中，player 可以是一个 Player、一个对类型 &Player 的引用，也可以是一个类型 Box<Player> 或 Rc<Player> 的智能指针。.location() 方法可以接收 player 的值或引用。相同的 .location() 语法适用于上述所有情形，因为 Rust 的 . 操作符会根据需求自动解引用 player 或借用一个对它的引用。

第三种语法用于调用静态方法，比如 Vec::new()：

```
let mut numbers = Vec::new(); // 静态方法调用
```

静态方法与非静态方法的区别与面向对象语言中一样：静态方法通过类型（如 Vec::new()）调用，而非静态方法通过值（如 my_vec.len()）调用。

自然，方法可以链式调用：

```
Iron::new(router).http("localhost:3000").unwrap();
```

Rust 语法有一个怪癖，即通常用于函数调用或方法调用的语法不能用于泛型 Vec<T>：

```
return Vec<i32>::with_capacity(1000); // 错误：要进行链式比较的

let ramp = (0 .. n).collect<Vec<i32>>(); // 同样的错误
```

问题在于表达式 < 是一个小于操作符。Rust 编译器对这种情况会建议使用 ::<T> 而不是 <T>：

```
return Vec::<i32>::with_capacity(1000); // 可以：使用::<

let ramp = (0 .. n).collect::<Vec<i32>>(); // 可以：使用::<
```

这样问题就解决了。Rust 社区亲切地称符号 ::<...> 为**极速鱼**（turbofish）。

此外，也可以省略类型参数，让 Rust 自己推断：

```
return Vec::with_capacity(10); // 可以：只要fn返回类型是Vec<i32>

let ramp: Vec<i32> = (0 .. n).collect(); // 可以：变量的类型已给定
```

只要可以推断出来，省略类型参数是推荐的做法。

6.9 字段与元素

结构体的字段使用类似的语法存取。元组的元素除了使用数值而非名字访问之外也一样：

```
game.black_pawns   // 结构体字段
coords.1           // 元组元素
```

如果 . 左侧的值是一个引用或智能指针类型，它就会跟方法调用一样自动解引用。

方括号用于访问数组、切片或向量中的元素：

```
pieces[i]     // 数组元素
```

方括号左侧的值会自动解引用。

类似下面这 3 个这样的表达式叫作**左值**（lvalue），因为它们会出现在赋值操作的左侧：

```
game.black_pawns = 0x00ff0000_00000000_u64;
coords.1 = 0;
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

当然，game、coords 和 pieces 必须声明为 mut 变量才行。

从数组或向量中提取切片很直观：

```
let second_half = &game_moves[midpoint .. end];
```

这里 game_moves 可以是数组、切片或向量。结果则是一个借用的切片，长度为 end - midpoint。game_moves 在 second_half 的生命期中是被借用的。

范围操作符 .. 允许省略两侧的操作数。根据操作数的个数，有可能产生 4 种不同类型的对象：

```
..      // RangeFull
a ..    // RangeFrom { start: a }
.. b    // RangeTo { end: b }
a .. b  // Range { start: a, end: b }
```

Rust 范围是**半开口**（half-open）的：包含起始值（如果有的话），不包含结尾值。范围 `0 .. 4` 包含数值 `0`、`1`、`2` 和 `3`。

只有包含起始值的范围才是可迭代的，因为循环必须从某个地方开始。但在数组切片中，以上 4 种形式都是有用的。如果省略范围的起始和结尾，则默认以数据的起始和结尾来切割。

因此，下面就是一个采用经典分治法实现快速排序的例子（部分代码）：

```
fn quicksort<T: Ord>(slice: &mut [T]) {
    if slice.len() <= 1 {
        return; // 没什么要排序
    }

    // 将切片分成前、后两部分
    let pivot_index = partition(slice);

    // 递归对slice的前半部分进行排序
    quicksort(&mut slice[.. pivot_index]);

    // 递归对slice的后半部分进行排序
    quicksort(&mut slice[pivot_index + 1 ..]);
}
```

6.10　引用操作符

取地址操作符 `&` 和 `&mut` 在第 5 章介绍过。

一元操作符 `*` 用于访问引用指向的值。如前所述，在使用 `.` 操作符时 Rust 会自动跟踪引用去访问字段或方法，因此 `*` 只在需要读或写该引用指向的整个值时才是必要的。

比如，有时候迭代器会产生引用，但程序需要的是下面的值：

```
let padovan: Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```

在这个例子中，`elem` 的类型是 `&u64`，因此 `*elem` 就是 `u64`。

6.11　算术、位、比较和逻辑操作符

Rust 的二元操作符跟许多其他语言中的一样。为节省时间，假设读者至少熟悉其中一种语言，这里只介绍 Rust 的不同之处。

Rust 有常用的算术操作符 `+`、`-`、`*`、`/` 和 `%`。第 3 章提到过，调试构建会检查出整数溢出并导致诧异。标准库为未检查的算术计算提供了类似 `a.wrapping_add(b)` 这样的方法。

即使在发布构建中，整数被零除也会触发诧异。整数有一个方法 `a.checked_div(b)`，其会返回一个 `Option`（如果 `b` 是 0 则为 `None`），并永远不会导致诧异。

一元 - 将数值取反。除了无符号整数，所有数值类型都支持它。没有一元 + 操作符。

```
println!("{}", -100);      // -100
println!("{}", -100u32);   // 错误：不能给u32类型应用一元-
println!("{}", +100);      // 错误：期待表达式，发现+
```

与 C 类似，`a % b` 计算除法的余数或模数。结果的符号与左手操作数相同。注意，`%` 既可以用于浮点数，也可以用于整数：

```
let x = 1234.567 % 10.0;   // 约等于4.567
```

Rust 也继承了 C 的按位整数操作符 `&`、`|`、`^`、`<<` 和 `>>`。不过，对于按位非，Rust 使用 `!` 而不是 `~`：

```
let hi: u8 = 0xe0;
let lo = !hi;  // 0x1f
```

这意味着对于整数 `n`，不能用 `!n` 表示“*n* 是 0”。为此，要写成 `n == 0`。

按位移动对于有符号整数始终是以符号位填充，对于无符号整数则始终是以零填充。因为 Rust 有无符号整数，所以它不需要 Java 的 `>>>`（无符号右移）操作符。

与 C 不一样，Rust 中位操作的优先级高于比较操作。因此，`x & BIT != 0` 的意思是 `(x & BIT) != 0`，通常这都是我们想要的。这比在 C 中的含义 `x & (BIT != 0)`（用于测试错位）更有用。

Rust 的比较操作符是 `==`、`!=`、`<`、`<=`、`>` 和 `>=`。被比较的两个值必须具有相同的类型。

Rust 还有两个短路逻辑操作符 `&&` 和 `||`。它们的操作数必须都是 `bool` 类型。

6.12　赋值

赋值操作符 `=` 用于把值赋给 `mut` 变量以及它们的字段或元素。但赋值在 Rust 中不像其他语言中那么常见，因为变量默认是不可修改的。

第 4 章提到过，赋值会**转移**非可复制类型的值，而不是隐式地复制它们。

Rust 也支持复合赋值：

```
total += item.price;
```

这相当于 `total = total + item.price;`。其他的操作符也支持 `-=`、`*=`，等等。完整的清单参见本章末尾的表 6-1。

与 C 不一样，Rust 不支持链式赋值，即不能通过 `a = b = 3` 把值 3 同时赋给 `a` 和 `b`。在 Rust 中需要这样连续赋值的情形少之又少。

Rust 没有 C 的递增操作符 `++` 和递减操作符 `--`。

6.13　类型转换

在 Rust 中，将一个值从一种类型转换为另一种类型通常需要显式转换。类型转换使用 `as` 关键字：

```
let x = 17;              // x的类型是i32
let index = x as usize;  // 转换为usize
```

以下是几种允许的类型转换。

- 数值可以从任何内置的数值类型转换为任意其他类型。

  整数转换为整数始终是意义明确的。转换为更窄的类型会导致截断。有符号整数转换为更宽的类型会以符号填充，无符号整数会以零填充，等等。简言之，没有什么意外。

  不过，截至本书撰写之时，把大浮点值转换为太小而无法表示它的整数类型会导致未定义行为，这即便在安全的 Rust 代码中也可能造成程序崩溃。这是编译器的一个 bug。

- `bool`、`char` 或类 C 的 `enum` 类型的值，可以转换为任何整数类型。（第 10 章将讨论枚举。）

  另一个方向的转换是不允许的，因为 `bool`、`char` 和 `enum` 类型对自己的值都有限制，都必须强制通过运行时检查。比如，禁止将 `u16` 转换为 `char` 是因为某些 `u16` 值（如 `0xd800`）对应的是 Unicode 代理码点，因此转换后得不到有效的 `char` 值。标准方法 `std::char::from_u32()` 会执行运行时检查且返回 `Option<char>`。不过，说来说去，这种转换实际上没有太大必要。我们一般会一次性转换整个字符串或流，而涉及 Unicode 文本的算法通常也不简单，最好还是通过库去做。

  有一个例外，就是 `u8` 可以转换为 `char`，因为从 0 到 255 的所有整数都是适合 `char` 存储的有效的 Unicode 码点。

- 有些涉及不安全指针类型的转换也是允许的。参见 21.7 节。

前面说过，转换**通常**需要显式进行。不过，有些涉及引用类型的转换非常简单直接，根本不需要显式进行。比如，把一个 `mut` 引用转换为非 `mut` 引用。

以下是另外一些比较重要的自动转换。

- `&String` 类型的值会自动转换为 `&str` 类型。
- `&Vec<i32>` 类型的值会自动转换为 `&[i32]`。
- `&Box<Chessboard>` 类型的值会自动转换为 `&Chessboard`。

这些转换被称为“解引用强制转换”（deref coercion），因为它们适应于实现了内置的 `Deref` 特型的类型。解引用强制转换的目的是让 `Box` 这种智能指针类型看起来尽可能像它后面的值。因为实现了 `Deref`，所以使用 `Box<Chessboard>` 多数情况下跟使用 `Chessboard` 没什么区别。

用户定义类型也可以实现 `Deref` 特型。如果你需要编写自己的智能指针类型，可以参考 13.5 节。

6.14　闭包

Rust 有**闭包**，即类似函数的轻量值。闭包通常由参数列表（在两条竖线中给出）和表达式组成。

```
let is_even = |x| x % 2 == 0;
```

Rust 推断参数类型和返回类型。当然也可以明确地写出来，就跟定义函数一样。如果指定了返回类型，那么闭包体必须是一个块，否则是个语法错误：

```
let is_even = |x: u64| -> bool x % 2 == 0; // 错误

let is_even = |x: u64| -> bool { x % 2 == 0 }; // 可以
```

调用闭包与调用函数的语法相同：

```
assert_eq!(is_even(14), true);
```

闭包是 Rust 最讨人喜欢的特性之一，关于它有很多东西可以讲。第 14 章将详细讨论。

6.15 优先级与关联性

表 6-1 总结了 Rust 表达式语法。操作符按照优先级从高到低的顺序给出。（与大多数编程语言一样，Rust 也有**操作符优先级**，其用于在表达式中包含多个毗连操作符时确定操作的顺序。比如，在 `limit < 2 * broom.size + 1` 中，`.` 操作符具有最高优先级，因此首先会读取该字段的值。）

表6-1：Rust表达式小结

表达式类型	示　例	相关特型
数组字面量	`[1, 2 , 3]`	
重复的数组字面量	`[0; 50]`	
元组	`(6, "crullers")`	
分组	`(2 + 2)`	
块	`{ f(); g() }`	
控制流表达式	`if ok { f() }`	
	`if ok { 1 } else { 0 }`	
	`if let Some(x) = f() { x } else { 0 }`	
	`match x { None => 0, _ => 1 }`	
	`for v in e { f(v); }`	`std::iter::IntoIterator`
	`while ok { ok = f(); }`	
	`while let Some(x) = it.next() { f(x); }`	
	`loop { next_event(); }`	
	`break`	
	`continue`	
	`return 0`	
宏调用	`println!("ok")`	
路径	`std::f64::consts::PI`	
结构体字面量	`Point {x: 0, y: 0}`	
元组字段存取	`pair.0`	`Deref`、`DerefMut`

（续）

表达式类型	示　　例	相关特型
结构体字段存取	`point.x`	`Deref`、`DerefMut`
方法调用	`point.translate(50, 50)`	`Deref`、`DerefMut`
函数调用	`stdin()`	`Fn(Arg0, ...) -> T`、 `FnMut(Arg0, ...) -> T`、 `FnOnce(Arg0, ...) -> T`
索引	`arr[0]`	`Index`、`IndexMut` `Deref`、`DerefMut`
错误检查	`create_dir("tmp")?`	
逻辑 / 按位非	`!ok`	`Not`
取反	`-num`	`Neg`
解引用	`*ptr`	`Deref`、`DerefMut`
借用	`&val`	
类型转换	`x as u32`	
乘	`n * 2`	`Mul`
除	`n / 2`	`Div`
取余（取模）	`n % 2`	`Rem`
加	`n + 1`	`Add`
减	`n - 1`	`Sub`
左移	`n << 1`	`Shl`
右移	`n >> 1`	`Shr`
按位与	`n & 1`	`BitAnd`
按位异或	`n ^ 1`	`BitXor`
按位或	`n | 1`	`BitOr`
小于	`n < 1`	`std::cmp::PartialOrd`
小于等于	`n <= 1`	`std::cmp::PartialOrd`
大于	`n > 1`	`std::cmp::PartialOrd`
大于等于	`n >= 1`	`std::cmp::PartialOrd`
等于	`n == 1`	`std::cmp::PartialEq`
不等于	`n != 1`	`std::cmp::PartialEq`
逻辑与	`x.ok && y.ok`	
逻辑或	`x.ok || backup.ok`	
范围	`start .. stop`	
赋值	`x = val`	
复合赋值	`x *= 1`	`MulAssign`
	`x /= 1`	`DivAssign`
	`x %= 1`	`RemAssign`
	`x += 1`	`AddAssign`
	`x -= 1`	`SubAssign`

（续）

表达式类型	示　　例	相关特型
	`x <<= 1`	`ShlAssign`
	`x >>= 1`	`ShrAssign`
	`x &= 1`	`BitAndAssign`
	`x ^= 1`	`BitXorAssign`
	`x |= 1`	`BitOrAssign`
闭包	`|x, y| x + y`	

所有这些操作符在链式操作时都具有左关联性。换句话说，类似 `a - b - c` 这样的链式操作会被分组为 `(a - b) - c`，而不是 `a - (b - c)`。可以像这样链式操作的都是意料之中的那些操作符：

```
* / % + - << >> & ^ | && || as
```

比较操作符、赋值操作符和范围操作符（`..`）压根就不能链式操作。

6.16　展望

表达式在我们看来是“可运行的代码”，它是 Rust 程序中可以编译为机器指令的部分。然而，表达式在整个语言中只占一小部分。

这在多数编程语言中也一样。程序的首要任务是运行，但运行并不是唯一的任务。程序必须相互通信，必须方便测试，必须有组织且灵活，这样才能不断发展壮大。程序必须能够与其他团队的代码和服务互操作。即便只是运行，像 Rust 这样静态类型语言的程序也需要更多工具去组织数据，而不仅仅是局限于元组和数组。

接下来，本书会用几章篇幅讨论相关的特性，比如赋予程序结构的模块和包，以及赋予数据结构的结构体和枚举。

不过，在此之前要先简单地介绍一下错误处理这个重要的话题。

第 7 章

错误处理

> 我早就知道无论我能活多久，这种事情迟早会发生。
>
> ——George Bernard Shaw 论死亡

Rust 中的错误处理非常特别，因此有必要单独用一章来介绍。不过，这里的思路并不难理解，只是可能比较新而已。本章介绍了 Rust 中两种不同的错误处理机制：诧异（panic）和 `Result`。

普通的错误使用 `Result` 处理。这些错误通常由程序外部的事情导致，比如输入错误、网络中断，或者权限问题。换句话说，这种错误由不得我们，即使写得再完美的程序也会不时地碰到。本章大多数篇幅会讨论这种错误处理。不过，我们会先讨论诧异，因为它相对简单。

诧异用于处理另一种错误，即**永远不该发生的错误**。

7.1　诧异

如果程序本身存在 bug，编译器就会诧异。比如：

- 越界访问数组
- 整数被零除
- 在值为 `None` 的 `Option` 上调用 `.unwrap()`
- 断言失败

（还有一个同名的宏，叫 `panic!()`，其用于你自己的代码在发现错误时主动触发诧异。`panic!()` 接受可选的 `println!()` 风格的参数，用于构建错误消息。）

毋庸讳言，以上情形的共性在于它们都由程序员的错误所导致。因此我们的经验是：“最

好不要诧异”。

可人哪有不犯错的呢。在这些错误不该发生却发生的时候，该怎么办？明显地，Rust 会给你一个选择：要么诧异发生时展开（unwind）栈，要么中止进程。展开是默认选项。

7.1.1 展开栈

当海盗抢劫之后分赃时，船长会先拿走一半，普通船员则按比例分另一半。（海盗不喜欢小数，因此若发现无法整除，得数就会四舍五入，最终多出来的部分会分给船上的鹦鹉。）

```
fn pirate_share(total: u64, crew_size: usize) -> u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

这个算法一直沿用了几个世纪，直到有一天，抢劫过后，船长发现自己是唯一的幸存者。如果给这个函数的 `crew_size` 参数传入 0，就会导致被零除。在 C++ 中，这属于未定义行为。而在 Rust 中，这会触发诧异，典型的处理过程如下。

- 在终端打印出错误消息。

  ```
  thread 'main' panicked at 'attempt to divide by zero', pirates.rs:3780
  note: Run with `RUST_BACKTRACE=1` for a backtrace.
  ```

  如果你像错误消息中所说的那样设置了 `RUST_BACKTRACE` 环境变量，那么 Rust 也会在此时将栈信息转存起来。

- 栈被展开。这非常像 C++ 的异常处理。

  当前函数使用的任何临时值、局部变量或参数都会按照它们创建的顺序被反向清除。清除意味着随之而来的清理：程序之前使用的任何 `String` 或 `Vec` 都会被释放，打开的 `File` 会被关闭，等等。用户定义的 `drop` 方法也会被调用，参见 13.1 节。对于 `pirate_share()` 这个特例而言，没什么要清理的。

  当前函数调用被清理之后，控制流交还至调用者，再以同样的方式清除其变量和参数。接下来是**那个**函数的调用者，总之沿栈逐层清理。

- 最后，线程退出。如果诧异线程是主线程，那么整个进程退出（退出码非 0）。

对于这个有秩序的过程而言，**诧异**这个词可能容易让人误解。诧异不是崩溃，也不是未定义行为。相反，诧异更像 Java 的 `RuntimeException` 或 C++ 的 `std::logic_error`。其行为是明确定义的，只是不应该发生而已。

诧异是安全的，它不违反 Rust 的任何安全规则。就算你设法在使用标准库方法时触发诧异，也永远不会在内存中导致悬空指针或初始化一半的值。关键在于 Rust 能够在问题扩大化**之前**捕获无效的数组访问或其他类似的错误。如果任由问题扩大化，那才会导致不安全，因此 Rust 会展开栈。不过进程的其他部分还可以继续运行。

诧异是线程级别的。一个线程诧异时，其他线程可以正常运行自己的业务逻辑。第 19 章将介绍父线程如何发现子线程中的诧异，并优雅地处理相应的错误。

还有一种方法可以**捕获**栈展开，允许诧异线程存活并继续运行。标准库函数 `std::panic::catch_unwind()` 就是为此准备的。本章不会介绍如何使用该函数，这里只想让读者知道 Rust 的测试套件在断言失败时使用了这个机制来恢复线程执行。（在编写通过 C 或 C++ 调用的 Rust 代码时，这个机制也是必需的，因为在非 Rust 代码中展开栈是未定义的行为，具体细节参见第 21 章。）

理想情况下，我们写的代码应该是完美、永不诧异的。可世间哪有完人？为保证程序更加健壮，可以使用线程和 `catch_unwind()` 来处理诧异。但有一点要注意，这些工具只能捕获展开栈的诧异，而并所有诧异都会展开栈。

7.1.2 中止进程

展开栈是默认的诧异行为，但在两种情况下 Rust 不会展开栈。

如果在 Rust 展开第一个函数之后的清理期间 `.drop()` 方法触发了第二个诧异，那么这个诧异会被认为是致使的。Rust 会停止展开并中止整个进程。

同样，Rust 的诧异行为是可自定义的。如果编译时加上 `-C panic=abort`，那么编译后程序中的**第一个**诧异就会立即中止进程。（编译时加上这个选项，Rust 则无须知道如何展开栈，因此能够减少编译后代码的大小。）

关于 Rust 中的诧异就介绍到这儿吧。没什么可说的了，因为常规的 Rust 代码没有义务处理诧异。即便使用线程或 `catch_unwind()`，你所有处理诧异的代码也很可能只集中在几个地方。指望程序中的每个函数都能预测并处理自己代码中的 bug 是不现实的。接下来看一下由其他因素导致的错误。

7.2 结果

Rust 没有异常。相反，函数执行失败可以通过一个返回类型来表示：

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

这个 `Result` 类型表示可能失败。调用 `get_weather()` 函数时，要么会返回一个**成功的结果** `Ok(weather)`，要么会返回一个**错误的结果** `Err(error_value)`。如果是前者，`weather` 就是一个新的 `WeatherReport` 值；如果是后者，`error_value` 就是一个解释出了什么错的 `io::Error` 值。

Rust 要求我们在调用这个函数时必须写一些错误处理逻辑。如果不对这个 `Result` **做点什么**，就拿不到 `WeatherReport`。而如果没有使用 `Result` 值，编译器就会给出警告。

第 10 章将介绍标准库对 `Result` 的定义，以及如何自定义类似的类型。接下来将采取“开清单”的方式，重点介绍如何使用 `Result` 来正确地处理错误。

7.2.1 捕获错误

处理 `Result` 最周全的方式其实在第 2 章已经展示过了，就是使用 `match` 表达式。

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {}", err);
        schedule_weather_retry();
    }
}
```

这是 Rust 的方式，相当于其他语言中的 `try/catch`。这是正面处理错误，不把错误抛给调用者的做法。

使用 `match` 还是有一点啰唆，因此 `Result<T, E>` 针对特定的常见情况提供了几种方法，其中每个方法的实现中都有一个 `match` 表达式。（要了解所有 `Result` 的方法，可以参考在线文档。这里列出来的只是其中最常用的。）

- **`result.is_ok()`** 和 **`result.is_err()`** 返回 `bool` 值，告诉我们 `result` 是成功的结果还是错误的结果。
- **`result.ok()`** 返回 `Option<T>` 类型的成功值（如果有的话）。如果 `result` 是一个成功的结果，就返回 `Some(success_value)`；否则，返回 `None`，而丢弃错误值。
- **`result.err()`** 返回 `Option<E>` 类型的错误值（如果有的话）。
- **`result.unwrap_or(fallback)`** 返回成功值，如果 `result` 是成功的结果的话。否则，它返回 `fallback`，丢弃错误值。

  ```
  // 对南加州比较靠谱的预测
  const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);

  // 如果可能，取得实时天气预报
  // 如果不行，以惯常的值作后备
  let report = get_weather(los_angeles).unwrap_or(THE_USUAL);
  display_weather(los_angeles, &report);
  ```

  这是对 `.ok()` 的一个完美替代，因为返回类型是 `T` 而非 `Option<T>`。当然，只有在存在适当后备值的情况下才可以使用这个方法。

- **`result.unwrap_or_else(fallback_fn)`** 是类似的，只是传入的不是后备值，而是一个函数或闭包。这个方法适合计算后备值如果用不上会造成浪费的情况。只有在返回错误结果时才会调用 `fallback_fn`。

  ```
  let report =
      get_weather(hometown)
      .unwrap_or_else(|_err| vague_prediction(hometown));
  ```

  （第 14 章将详细介绍闭包。）

- **`result.unwrap()`** 也会返回成功值（如果 `result` 是成功的结果的话）。不过，如果 `result` 是错误的结果，这个方法则会诧异。这个方法也有它的用途，稍后再谈。
- **`result.expect(message)`** 与 `.unwrap()` 相同，只不过你需要自己提供诧异时打印到控制台的消息。

最后，再看两个借用 `Result` 中值的引用的方法。

- **`result.as_ref()`** 将 `Result<T, E>` 转换为 `Result<&T, &E>`，即借用现有 `result` 中成功或错误值的引用。
- **`result.as_mut()`** 也一样，只是借用了可修改引用。返回类型为 `Result<&mut T, &mut E>`。

这两个方法之所以有用，是因为前面列出的方法中，除了 `.is_ok()` 和 `.is_err()` 之外，其他方法都会**用掉**调用它们的 `result` 值。换句话说，它们通过 `self` 参数得到了 `result` 的值。有时候，访问结果中的数据又不毁坏它是很方便的，而这正是 `.as_ref()` 和 `.as_mut()` 派上用场的时候。比如，假设你想调用 `result.ok()`，但需要 `result` 原封不动。那么可以写成 `result.as_ref().ok()`，这样就只会借用 `result`，返回一个 `Option<&T>` 而非 `Option<T>`。

7.2.2 结果类型别名

在看 Rust 文档时，有时候你可能会发现有些代码省略了 `Result` 的错误类型：

```
fn remove_file(path: &Path) -> Result<()>
```

这其实是使用了 `Result` 类型别名。

类型别名有点类似类型名的简写。模块经常会定义 `Result` 类型别名，以避免模块中的每个函数将一个错误类型重复写很多遍。比如，标准库的 `std::io` 模块中有如下代码：

```
pub type Result<T> = result::Result<T, Error>;
```

这定义了一个公有类型 `std::io::Result<T>`，它以硬编码的 `std::io::Error` 作为错误类型的 `Result<T, E>` 的别名。实际使用中，如果你写了 `use std::io`，那么 Rust 就会认为 `io::Result<String>` 是 `Result<String, io::Error>` 的简写。

因此，当在线文档中出现类似 `Result<()>` 这样的内容时，你可以单击标识符 `Result`，查看这里使用的是什么类型的别名，进而得出错误类型。实践中，通常可以根据上下文判断错误类型。

7.2.3 打印错误

有时候处理错误的唯一方式可能就是把错误转存到终端，然后继续执行。前面已经展示了这样处理的一个例子：

```
println!("error querying the weather: {}", err);
```

标准库定义了几种错误类型，名字都很无聊：`std::io::Error`、`std::fmt::Error`、`std::str::Utf8Error`，等等。这些错误都实现了一个公共接口，即 `std::error::Error` 特型。这意味着它们具有以下特点。

- 它们都可以使用 `println!()` 来打印。打印错误时以 `{}` 作为格式描述符通常只会显示简略的错误消息。或者，可以选择使用 `{:?}` 作为格式描述符，此时会看到错误的 `Debug` 版。虽然看起来有点乱，但会包含额外的技术细节。

```
// println!("error: {}", err);的结果
error: failed to lookup address information: No address associated with
hostname

// println!("error: {:?}", err);的结果
error: Error { repr: Custom(Custom { kind: Other, error: StringError(
"failed to lookup address information: No address associated with
hostname") }) }
```

- **`err.description()`** 返回 `&str` 类型的错误消息。
- **`err.cause()`** 返回一个 `Option<&Error>`，这是触发 `err` 的底层错误（如果有的话）。

  比如，某个网络错误可能导致银行交易失败，而交易失败可能导致你的游艇被收回。如果 `err.description()` 是 `"boat was repossessed"`（游艇被收回），那么 `err.cause()` 可能会返回一个关于交易失败的错误。这个错误的 `.description()` 可能是 `"failed to transfer $300 to United Yacht Supply"`（给 United Yacht Supply 转账 300 美元失败），而它的 `.cause()` 可能是一个 `io::Error`，包含导致所有这些麻烦的特定网络中断的相关信息。这里第三个错误就是根原因，因此它的 `.cause()` 方法会返回 `None`。

  因为标准库只包含非常底层的特性，所以标准库函数的错误通常都是 `None`。

打印错误消息不一定会打印出错误原因。如果你确实需要打印所有可用信息，可以使用这个函数：

```
use std::error::Error;
use std::io::{Write, stderr};

/// 把错误消息转存到stderr
///
/// 如果在构建当前错误消息或写入stderr时发生另一个错误，则忽略该错误
fn print_error(mut err: &Error) {
    let _ = writeln!(stderr(), "error: {}", err);
    while let Some(cause) = err.cause() {
        let _ = writeln!(stderr(), "caused by: {}", cause);
        err = cause;
    }
}
```

标准库的错误类型不包含栈追踪信息，但使用 `error-chain` 包可以方便地定义自己的错误类型，以支持在创建时获取栈追踪信息。这个包使用 `backtrace` 捕获栈信息。

7.2.4 传播错误

在尝试可能出错的操作时，大多数情况下我们并不希望立即捕获和处理错误。如果在每个可能出错的地方都写上 10 行 `match` 语句，那就有点过分了。

这时候，我们通常会希望调用者来处理错误。换句话说，我们希望错误可以沿调用栈向上**传播**。

Rust 的 `?` 操作符可以传播错误。可以在任何产生 `Result` 的表达式后面添加 `?`，例如在函数调用的结果后面：

```
let weather = get_weather(hometown)?;
```

? 操作符的行为取决于这个函数是返回一个成功结果，还是返回一个错误结果。

- 如果是成功结果，那么它会打开 `Result` 并取出其中的成功值。这里 `weather` 的类型不是 `Result<WeatherReport, io::Error>`，而是简单的 `WeatherReport`。
- 如果是错误结果，那么它会立即从闭合函数中返回，将错误结果沿调用链向上传播。为确保传播成功，只能对返回类型为 `Result` 的函数使用 `?`。

? 操作符并不神秘。同样的操作使用一个 `match` 表达式也能实现，只不过太长了：

```
let weather = match get_weather(hometown) {
    Ok(success_value) => success_value,
    Err(err) => return Err(err)
};
```

`match` 表达式与 `?` 的唯一区别在于类型和转换方面的细节。下一节会详细探讨这个话题。

在较早的代码中，你可能会看到 `try!()` 宏，这是在 Rust 1.13 中引入 `?` 操作符之前用于传播错误的典型方式。

```
let weather = try!(get_weather(hometown));
```

`try!()` 宏会扩展为一个类似上面这样的 `match` 表达式。

人们很容易忽视程序出错的可能性，特别是在与操作系统打交道的代码中，出错的概率就更高了。因此，甚至有时候你会发现在一个函数的每一行的末尾都会跟着一个 `?` 操作符：

```
use std::fs;
use std::io;
use std::path::Path;

fn move_all(src: &Path, dst: &Path) -> io::Result<()> {
    for entry_result in src.read_dir()? {  // 打开dir可能失败
        let entry = entry_result?;         // 读取dir可能失败
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?; // 重命名可能失败
    }
    Ok(()) // 哇哦!
}
```

7.2.5　处理多种错误类型

我们经常会碰到多种错误同时出现的情况。假设就是从文本文件中读取数值。

```
use std::io::{self, BufRead};

/// 从文本文件中读取整数
/// 这个文件中的每一行应该都有一个数值
fn read_numbers(file: &mut BufRead) -> Result<Vec<i64>, io::Error> {
    let mut numbers = vec![];
    for line_result in file.lines() {
        let line = line_result?;          // 读取行的内容可能失败
        numbers.push(line.parse()?);     // 解析整数有可能失败
```

```
    }
    Ok(numbers)
}
```

Rust 会报一个编译错误：

```
numbers.push(line.parse()?);     // 解析整数有可能失败
             ^^^^^^^^^^^^^ the trait `std::convert::From<std::num::ParseIntError>`
                           is not implemented for `std::io::Error`
```

关于报错信息中提到的特型（trait），可能等到看过第 11 章后会更容易理解。眼下，只要知道 Rust 在抱怨它不能把 `std::num::ParseIntError` 值转换为 `std::io::Error` 就可以了。

这里的问题是从文件中读取一行内容并解析为整数时会产生两种不同的潜在错误类型，其中，`line_result` 的类型是 `Result<String, std::io::Error>`，而 `line.parse()` 的类型是 `Result<i64, std::num::ParseIntError>`。函数 `read_numbers()` 的返回类型只能容纳 `io::Error` 错误。为此，Rust 会尝试把 `ParseIntError` 转换为 `io::Error`，但这种转换的可能性不存在，因此我们就得到了一个类型错误。

处理这种情况的方法不止一种。比如，第 2 章用到的为曼德布洛特集合创建图像文件的 `image` 包定义了自己的错误类型 `ImageError`，并实现了从 `io::Error` 及其他几种错误类型到 `ImageError` 的转换。如果你想采取这个路线，那么可以试一试前面提到的 `error-chain` 包，它可以帮你只用几行代码就定义出灵活的错误类型。

另一个更简单的方法是使用 Rust 内置的特性。所有标准库的错误类型都可以转换为 `Box<std::error::Error>` 类型，其含义为“任何错误”。因此，处理多种错误类型的一个简单方案就是定义如下类型别名：

```
type GenError = Box<std::error::Error>;
type GenResult<T> = Result<T, GenError>;
```

然后，把 `read_numbers()` 的返回类型改为 `GenResult<Vec<i64>>`。这么一改，函数编译就通过了。`?` 操作符会根据需求自动将任意错误类型转换为 `GenError`。

顺便提一下，`?` 操作符使用了一个标准库方法来实现上述自动转换。实际上你自己也可以使用该方法。要把任意错误转换为 `GenError` 类型，可以调用 `GenError::from()`：

```
let io_error = io::Error::new(         // 创建自己的io::Error
    io::ErrorKind::Other, "timed out");
return Err(GenError::from(io_error)); // 手工转换为GenError
```

第 13 章将介绍 `From` 特型及其 `from()` 方法。

使用 `GenError` 的缺点在于返回类型不再精确地传达调用者可以预测的错误类型。调用者必须做好各方面准备。

如果调用的函数返回 `GenResult`，但你只想处理一种特定的错误，而让其他所有错误传播出去，可以使用泛型方法 `error.downcast_ref::<ErrorType>()`。**如果**恰好是你想要的那个错误类型，那么该方法会借用对这个错误的引用：

```
loop {
    match compile_project() {
        Ok(()) => return Ok(()),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<MissingSemicolonError>() {
                insert_semicolon_in_source_code(mse.file(), mse.line())?;
                continue; // 再试!
            }
            return Err(err);
        }
    }
}
```

很多语言为完成此类操作提供了内置语法，实际上完全没有必要。Rust 只是专门为此提供了一个方法。

7.2.6　处理“不会发生”的错误

某些情况下，我们就**知道**某个错误不会发生。比如，假设我们正在写一段代码来解析一个配置文件，写着写着发现文件里接下来的数据是一个数字字符串：

```
if next_char.is_digit(10) {
    let start = current_index;
    current_index = skip_digits(&line, current_index);
    let digits = &line[start..current_index];
    ...
```

我们想把这个数字字符串转换成实际的数值。有一种标准的方法专门做这件事：

```
let num = digits.parse::<u64>();
```

那么问题来了：`str.parse::<u64>()` 方法并不返回 `u64`，而是返回一个 `Result`。转换可能失败，因为有些字符串不是数值：

```
"bleen".parse::<u64>() // ParseIntError：无效数字
```

但在这里，我们恰好知道 `digits` 完全由数字组成。那么应该做点什么？

如果我们写的代码返回 `GenResult`，那可以附加一个 `?`，然后就没事了。否则，就会出现要为不会发生的错误编写错误处理代码的恼人局面。最好的办法是使用 `.unwrap()`，也就是前面提到过的 `Result` 的一个方法。

```
let num = digits.parse::<u64>().unwrap();
```

这样跟使用 `?` 差不多，只不过在判断出错时，也就是会发生错误时，代码会诧异。

事实上，在特殊情况下，我们确实可能判断出错。如果输入包含足够长的数字字符串，那么转换后的数值会因为太大而难以放到 `u64` 里。

```
"99999999999999999999".parse::<u64>() // 溢出错误
```

在这种特殊情况下使用 `.unwrap()` 就成了一个 bug。而有问题的输入不应该导致诧异。

话虽如此，但确实存在 `Result` 值真的不能是错误的这种情况。比如，在第 18 章中，

Write 特型为文本和二进制输出定义了一组公共方法（.write() 等）。所有这些方法都返回 io::Result，如果你恰好要写入一个 Vec<u8>，那就不能失败。此时，使用 .unwrap() 或 .expect(message) 来省略 Result 是可以接受的。

如果错误代表的是一个非常严格或奇怪的条件，而你希望在出错时诧异，那也可以使用这些方法。

```
fn print_file_age(filename: &Path, last_modified: SystemTime) {
    let age = last_modified.elapsed().expect("system clock drift");
    ...
}
```

在这里，.elapsed() 方法只有当系统时间早于文件创建时间时才会失败。如果文件是最近刚创建的，而系统时钟在程序运行时被向后调整过，那就有可能满足失败条件。根据使用代码的方式，在这种情况下触发诧异而不是处理错误或将错误传播给调用者，倒不失为一种合理的做法。

7.2.7　忽略错误

偶尔，我们也会希望忽略某个错误。比如，在 print_error() 函数中，我们必须处理一种不太可能的情况，即打印错误时触发另一个错误。这是有可能的，比如，把 stderr 通过管道发送到另一个进程，而该进程被杀死了。因为对这种类型的错误，我们几乎没什么可做的，所以想忽略它。但 Rust 编译器会警告说存在未使用的 Result 值：

```
writeln!(stderr(), "error: {}", err); // 警告：未使用的结果
```

惯用法 let _ = ... 可以用来禁止这种警告：

```
let _ = writeln!(stderr(), "error: {}", err); // 没问题，忽略结果
```

7.2.8　在main()中处理错误

在产生 Result 的大多数情况下，让错误冒泡到调用者是正确的做法。这也是 ? 在 Rust 中只有一个字符的原因。前面我们也看到过，在某些程序里很多行末尾有它。

但是，如果错误的传播路径足够长，最终抵达了 main()，就不能继续传播了。main() 不能使用 ?，因为它的返回类型不是 Result。

```
fn main() {
    calculate_tides()?;    // 错误：不能把麻烦再向外传播了
}
```

在 main() 中处理错误的最简单方式是使用 .expect()。

```
fn main() {
    calculate_tides().expect("error");   // 麻烦到此为止
}
```

如果 calculate_tides() 返回一个错误结果，.expect() 方法则会诧异。主线程中的诧异会打印错误消息，然后以一个非零退出码退出。这基本上可以算是我们想要的结果。对小程

序来说，我们一直都是这么干的。小程序是一切程序的开始。

不过，错误消息还是有点让人紧张：

```
$ tidecalc --planet mercury
thread 'main' panicked at 'error: "moon not found"', /buildslave/rust-buildbot/s
lave/nightly-dist-rustc-linux/build/src/libcore/result.rs:837
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

错误消息混在了一堆文字中间。同样，在这种情况下 `RUST_BACKTRACE=1` 不是个好主意。最好还是自己来打印错误消息：

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

以上代码使用了 `if let` 表达式，只在调用 `calculate_tides()` 返回错误结果时打印错误消息。有关 `if let` 表达式的详细解释，参见第 10 章。而 `print_error` 函数的定义可以在 7.2.3 节找到。

现在，输出简洁明了：

```
$ tidecalc --planet mercury
error: moon not found
```

7.2.9 声明自定义错误类型

假设你在编写一个新的 JSON 解析器，而且想让它拥有自己的错误类型。（本书至今尚未介绍用户定义类型，接下来几章会介绍。错误类型比较有用，这里先大概看一看。）

以下大概是你需要写的最低限度的代码：

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

这个结构体可以通过 `json::error::JsonError` 调用，在想要创建一个这种类型的错误时，可以这样写：

```
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

没问题。不过，如果你想让这个错误类型的行为接近标准错误类型，正如你的库用户所期待的那样，那么还要多写一些逻辑：

```
use std;
use std::fmt;

// 错误应该可以打印出来
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({}:{})", self.message, self.line, self.column)
    }
}

// 错误应该实现std::error::Error特型
impl std::error::Error for JsonError {
    fn description(&self) -> &str {
        &self.message
    }
}
```

同样，这里的 `impl` 关键字、`self` 还有其他细节，都会在接下来的几章里介绍。

7.2.10 为什么是结果

至此，我们应该可以理解 Rust 选择 `Result` 而不是异常的用意所在了。以下是设计的要点。

- Rust 要求程序员在所有可能发生错误的地方做出某种决定，并记录在代码中。这没有任何问题，毕竟人很容易因为疏忽而忘记处理错误。
- 最常见的决定是让错误向外传播，而这只需要写一个字符 `?`。这样错误就不会像 C 和 Go 中那样弄乱你的代码了。而且传播路径是可见的：就算是一堆代码，只需用眼睛扫一下，就能知道错误会从哪些地方传播出来。
- 由于每个函数的返回类型都包含了出错的可能，因此哪个函数可以失败，哪个函数不能失败非常清晰。如果把一个函数修改为可以失败，那就要修改其返回类型，而编译器会指导你更新该函数的下游调用者。
- Rust 检查是否使用了 `Result` 值，因此不可能静默传递某个错误（这是 C 中的一个常见错误）。
- `Result` 也是一个数据类型，所以在同一个集合中存储成功和错误结果很简单。因而对部分成功的结果建模也很容易。比如，你要编写一个从文本文件中加载几百万条记录的程序，那你必须有可以应对大多数情况下成功、少数情况下失败的方案。此时就可以在内存中通过一个 `Result` 向量来表示这种情况。

如此设计的代价，就是你会发现自己在 Rust 中要比在其他语言中花更多时间来思考和权衡错误处理。跟其他很多方面一样，Rust 对待错误处理的态度比你过去习惯的要稍微严格一些。对于系统编程来说，这样是值得的。

第 8 章

包和模块

> 这是对 Rust 主题的一个小备注：系统编程语言也可以有这么灵活的机制。
>
> ——Robert O'Callahan，"Random Thoughts on Rust: Crates.io and IDEs"

假设我们要写一个程序来模拟蕨类植物从细胞直到长大的生成过程。而这个程序，也与蕨类植物相似，一开始代码非常简单，可能所有代码都写在一个文件里（就是想法的孢子）。随着细胞不断分裂，蕨类植物将开始长出内部结构。不同的分支各有不同的用途。于是程序也会发展出多个文件，最终遍布整整一个目录树。某一天，这个程序可能会成为整个软件生态系统的重要组成部分。

本章介绍了 Rust 实现代码组织的相关特性：包和模块。具体内容将涉及项目成长过程中可能遇到的方方面面，比如如何为 Rust 代码编写文档、如何测试、如何屏蔽不需要的编译警告、如何使用 Cargo 管理项目依赖和进行版本控制、如何在 crates.io 上发布开源库，等等。

8.1 包

Rust 程序由**包**组成。每个包都是一个 Rust 项目，包含一个独立的库或可执行文件的全部源代码，以及相关的测试、示例、工具、配置和其他东西。对我们要实现的蕨类植物模拟程序而言，可能需要用到提供 3D、生物信息和并行计算功能等的第三方库。这些库都以包的形式发布，如图 8-1 所示。

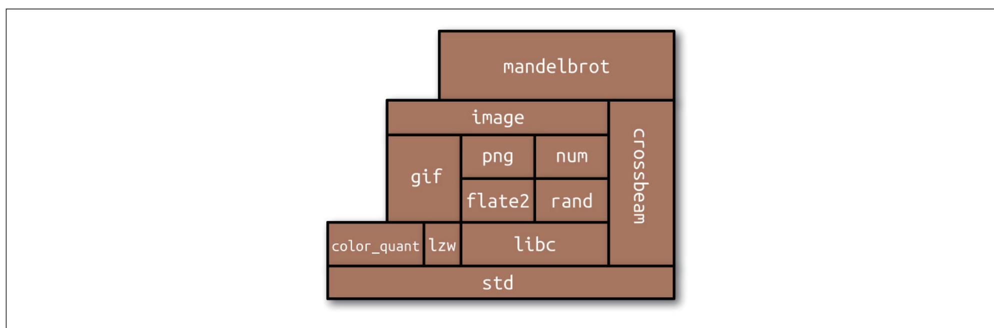


图 8-1：一个包及其依赖

要了解包是什么，以及它们如何协作，最简单的办法就是对一个使用了依赖的已有项目运行 `cargo build` 命令，同时加上 `--verbose` 标记。我们用 2.6.6 节“并发的曼德布洛特程序”作为例子，结果如下所示：

```
$ cd mandelbrot
$ cargo clean    # 删除之前编译的代码
$ cargo build --verbose
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading image v0.6.1
 Downloading crossbeam v0.2.9
 Downloading gif v0.7.0
 Downloading png v0.4.2

 ... (downloading and compiling many more crates)

   Compiling png v0.4.2
     Running `rustc .../png-0.4.2/src/lib.rs
          --crate-name png
          --crate-type lib
          --extern num=.../libnum-a2e6e61627ca7fe5.rlib
          --extern inflate=.../libinflate-331fc425bf167339.rlib
          --extern flate2=.../libflate2-857dff75f2932d8a.rlib
         ...`
   Compiling image v0.6.1
     Running `rustc .../image-0.6.1/./src/lib.rs
          --crate-name image
          --crate-type lib
          --extern png=.../libpng-16c24f58491a5853.rlib
          ...`
   Compiling mandelbrot v0.1.0 (file://.../mandelbrot)
     Running `rustc src/main.rs
          --crate-name mandelbrot
          --crate-type bin
          --extern crossbeam=.../libcrossbeam-ba292320058da7df.rlib
          --extern image=.../libimage-254ec48c8f0684f2.rlib
          ...`
$
```

为增强可读性，以上代码对 `rustc` 命令行做了格式化，而且删除了很多与讨论无关的编译

器选项，代之以省略号（`...`）。

有读者大概还记得，在那个例子完成时，曼德布洛特程序的 main.rs 文件中有 3 个 `extern crate` 声明：

```
extern crate num;
extern crate image;
extern crate crossbeam;
```

这几行声明只是告诉 Rust，`num`、`image` 和 `crossbeam` 都是外部库，并不是曼德布洛特程序本身的代码。

同时，在 Cargo.toml 文件里，我们也为每个包指定了对应的版本：

```
[dependencies]
num = "0.1.27"
image = "0.6.1"
crossbeam = "0.2.8"
```

此处**依赖**指的是当前项目用到的其他包，也就是我们要依赖的代码。这些包在以 Rust 为开源包搭建的网站 crates.io 上都能找到。比如，在浏览器中打开 crates.io，然后搜索“image”，就可以找到 `image` 库。crates.io 上每个包的页面都有链接指向相关文档和源代码，以及像 `image = "0.6.1"` 这样的可以直接复制到 Cargo.toml 文件中的一行配置。这里的版本号就是在写程序时这 3 个包的最新版本。

Cargo 的输出讲述了这里发生的一切。在运行 `cargo build` 时，Cargo 首先从 crates.io 上下载了这 3 个包指定版本的源代码。然后，它读取这些包的 Cargo.toml 文件，并下载**它们的**依赖，如此递归下去。比如，0.6.1 版 `image` 包源代码的 Cargo.toml 文件里列出了以下依赖：

```
[dependencies]
byteorder = "0.4.0"
num = "0.1.27"
enum_primitive = "0.1.0"
glob = "0.2.10"
```

看到这些，Cargo 就知道使用 `image` 前必须先取得这些包。稍后，我们会介绍如何告诉 Cargo 从 Git 存储库或本地文件系统而不是 crates.io 上取得源代码。

一旦取得了所有的源代码，Cargo 就会编译所有包。它会对项目依赖的每个包都运行一次 `rustc`，也就是 Rust 编译器。在编译第三方库时，Cargo 会使用 `--crate-type lib` 选项。这会告诉 `rustc` 不要去找 `main()` 函数，而是生成一个 .rlib 文件，其中包含编译后的代码，这些代码的格式可供之后的 `rustc` 命令用作输入。在编译程序时，Cargo 会使用 `--crate-type bin` 选项，编译结果将是一个针对目标平台的二进制可执行文件，比如在 Windows 上就是 mandelbrot.exe。

运行每个 `rustc` 命令时，Cargo 会通过 `--extern` 选项给出当前包用到的每个库的文件名。这样，当 `rustc` 看到 `extern crate crossbeam;` 这行代码时，它就知道到磁盘的什么位置去找这个库编译后的代码了。Rust 编译器需要访问 .rlib 文件，因为其中包含第三方库编译后的代码。Rust 会将这些代码静态链接到最终的可执行文件上。.rlib 文件也包含类型信息，Rust 可以据此检查我们代码中用到的库特性确实在对应的包里存在，从而保证正确地使用

它们。这个文件里还包含包的公共内联函数、泛型和宏的一个副本，这些特性直到 Rust 遇到调用它们的代码时才会编译为机器码。

`cargo build` 支持很多选项，其中大多数本书将不作介绍。不过，这里还是要提一下：`cargo build --release` 产生优化的代码。优化的代码运行更快，但编译时间比较长，而且不会检查整数溢出，还会跳过 `debug_assert!()` 断言，另外它们针对诧异生成的栈追踪信息一般不太可靠。

构建分析

表 8-1 总结了可以放到 Cargo.toml 文件中的几种配置，它们会影响 `cargo` 生成的 `rustc` 命令行。

表8-1：构建分析

命　令　行	Cargo.toml使用的区块
`cargo build`	`[profile.dev]`
`cargo build --release`	`[profile.release]`
`cargo test`	`[profile.test]`

默认执行的构建通常就可以了，但有时候也许需要使用分析程序，即一个可以度量你的程序是哪部分花了太多 CPU 时间的程序。要从分析程序获得最全面的数据，需要同时启用优化（通常只在发布构建时启用）和调试（通常只在调试构建时启用）符号（symbol）。如果想两者都启用，必须在 Cargo.toml 中添加如下代码：

```
[profile.release]
debug = true   # 在发布构建中启用调试标记
```

这里的 `debug` 设置控制 `rustc` 中的 `-g` 选项。有了这个配置，再执行 `cargo build --release`，就可以得到一个带有调试符号的二进制文件。优化设置不受影响。

Cargo 文档中给出了可以调整的其他很多设置。

8.2　模块

模块既是 Rust 的命名空间，也是函数、类型、常量等构成 Rust 程序或库的容器。包主要解决项目间代码共享的问题，而模块主要解决**项目内**代码组织的问题。下面就是一个模块：

```
mod spores {
    use cells::Cell;

    /// 细胞（cell）由成熟的蕨类产生。在蕨类的生命期内，它会随风传播。
    /// 孢子会成长为原叶体，即一个完全独立的有机体，最大有5毫米。原叶体
    /// 会产生接合子，接合子则会长成新的蕨类（植物的性别很复杂）
    pub struct Spore {
        ...
    }
```

```
    /// 模拟细胞分裂产生孢子
    pub fn produce_spore(factory: &mut Sporangium) -> Spore {
        ...
    }

    /// 混合基因以备细胞分裂（细胞间期）
    fn recombine(parent: &mut Cell) {
        ...
    }

    ...
}
```

模块是**特性项**（item）的集合，比如这个例子中的结构体 `Spore` 和两个函数。关键字 `pub` 用于标记公有或公开的特性项，以便在模块外部能够访问它。任何没有标记为 `pub` 的特性项都是模块私有的。

```
let s = spores::produce_spore(&mut factory); // 可以

spores::recombine(&mut cell); // 错误：recombine是私有的
```

模块可以嵌套，包含一系列子模块的模块也是很常见的：

```
mod plant_structures {
    pub mod roots {
        ...
    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}
```

以这种代码组织方式当然可以写出完整的程序，把大量代码和层层嵌套的模块都放在一个源文件里。但这样组织代码显然不够灵活，因此还有更好的方式。

8.2.1 把模块写在单独的文件中

模块也可以这样写：

```
mod spores;
```

前面，我们的代码中也写出了 `spores` 模块的代码体，即包含在花括号中的内容。这里只是告诉 Rust 编译器，`spores` 模块保存在一个单独的名叫 spores.rs 的文件里：

```
// spores.rs

/// 细胞（cell）由成熟的蕨类产生……
pub struct Spore {
    ...
}
```

```
/// 模拟细胞分裂产生孢子
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}

/// 混合基因以备细胞分裂（细胞间期）
fn recombine(parent: &mut Cell) {
    ...
}
```

spores.rs 只包含构成模块的特性项。换句话说，包含模块的文件里不再需要任何多余的代码来声明这是一个模块。

写在文件中的 `spores` 模块与前面介绍的 `spores` 模块的唯一区别是代码保存在了不同的地方。关于公有还是私有的声明规则在两种写法下都适用。事实上，就算模块写在了单独的文件中，Rust 也不会单独编译模块。构建 Rust 包时，会重新编译包中的所有模块。

模块也可以有自己的目录。Rust 在看到 `mod spores;` 时，既会检查是否存在 spores.rs 文件，也会检查是否存在 spores/mod.rs 文件。如果两个文件都存在，或者都不存在，就会报错。这个例子使用的是 spores.rs 文件，因为 `spores` 模块没有任何子模块。再考虑一下前面写过的 `plant_structures` 模块。如果想把这个模块还有它的 3 个子模块都保存在各自的文件中，那项目的目录结构就会变成这样：

```
fern_sim/
├── Cargo.toml
└── src/
    ├── main.rs
    ├── spores.rs
    └── plant_structures/
        ├── mod.rs
        ├── leaves.rs
        ├── roots.rs
        └── stems.rs
```

在 main.rs 中声明 `plant_structures` 模块：

```
pub mod plant_structures;
```

这样 Rust 就会去加载 plant_structures/mod.rs，而在这个文件中，又会声明 3 个子模块：

```
// 在plant_structures/mod.rs中
pub mod roots;
pub mod stems;
pub mod leaves;
```

这 3 个模块的内容保存在名为 leaves.rs、roots.rs 和 stems.rs 的单独文件中，与 mod.rs 一起位于 plant_structrues 目录下。

8.2.2 路径和导入

操作符 `::` 用于访问模块的特性。项目中任何地方的代码都可以通过写出其**绝对路径**来引用标准库特性：

```
if s1 > s2 {
    ::std::mem::swap(&mut s1, &mut s2);
}
```

函数名 `::std::mem::swap` 是一个绝对路径，因为它以双冒号开头。路径 `::std` 引用标准库的顶级模块，而 `::std::mem` 引用标准库的子模块，相应地，`::std::mem::swap` 则引用该模块中的一个公有函数。

如果想写个圆或字典，可以写 `::std::f64::consts::PI` 或 `::std::collections::HashMap::new`。但每次都这么写既单调乏味，也不好认读。这时候可以把要使用的特性**导入**模块中：

```
use std::mem;

if s1 > s2 {
    mem::swap(&mut s1, &mut s2);
}
```

这里的 `use` 声明会让 `mem` 在整个代码块或者整个模块中成为 `::std::mem` 的一个局部别名。而 `use` 声明中的路径会自动转换成绝对路径，因此不需要写出前导的 `::`。

可以用 `user std::mem::swap;` 声明来导入 `swap` 函数本身，而不是 `mem` 模块。然而，上面的导入方式通常被认为是最佳方式：导入类型、特性和模块（如 `std::mem`），然后再使用相对路径访问其中的函数、常量及其他成员。

可以一次性导入多个名字：

```
use std::collections::{HashMap, HashSet}; // 同时导入两个模块

use std::io::prelude::*; // 导入所有模块
```

以上代码是对以下代码的简写：

```
use std::collections::HashMap;
use std::collections::HashSet;

// std::io::prelude中的所有公有特性项：
use std::io::prelude::Read;
use std::io::prelude::Write;
use std::io::prelude::BufRead;
use std::io::prelude::Seek;
```

模块**不会**自动从自己的父模块继承名字。比如，假设在一个父模块 proteins/mod.rs 中有如下声明：

```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

然后其子模块 synthesis.rs 中的代码不会自动看到类型 `AminoAcid`：

```
// proteins/synthesis.rs
pub fn synthesize(seq: &[AminoAcid]) // 错误：找不到类型AminoAcid
    ...
```

实际上，每个“白手起家”的模块都必须导入自己要使用的名字：

```
// proteins/synthesis.rs
use super::AminoAcid; // 从父模块中明确导入

pub fn synthesize(seq: &[AminoAcid]) // 可以
    ...
```

关键字 `super` 在导入声明中有特殊含义：它是父模块的一个别名。类似地，`self` 则是当前模块的一个别名。

```
// in proteins/mod.rs

// 从一个子模块中导入
use self::synthesis::synthesize;

// 从一个枚举中导入名字，
// 这样就可以用Lys而不是AminoAcid::Lys来表示赖氨酸
use self::AminoAcid::*;
```

虽然导入声明中的路径默认被当成绝对路径，但使用 `self` 和 `super` 可以改变配置，实现从相对路径导入。

（当然，这里 `AminoAcid` 的例子确实违背了前面提到的只导入类型、特型和模块的代码编写风格。假如程序中包含很长的氨基酸序列，那么就可以适用奥威尔第六法则："当必须避免说出粗鄙的话时，应打破这些法则。"）

子模块可以访问其父模块中的私有特性项，但必须通过名字导入每一项。使用 `super::*;` 只会导入那些被标记为 `pub` 的特性项。

模块与文件不是一码事，但 Unix 文件系统的文件和目录结构与模块具有天然的对应关系。`use` 关键字创建别名，就像 `ln` 命令创建链接一样。路径跟文件名一样，有绝对和相对两种形式。`self` 和 `super` 与特殊目录 `.` 和 `..` 类似。把另一个包的根模块移植到项目中的 `extern crate`，则很像是挂载一个文件系统。

8.2.3　标准前置模块

前面刚刚讨论导入名字的时候提到了每个"白手起家"的模块。实际上，这些模块并非那么"一穷二白"。

首先，标准库 `std` 会自动链接到每个项目。这就像 lib.rs 或 main.rs 中包含了一个引用它的不可见的声明：

```
extern crate std;
```

其次，一些特别常用的名字（比如 `Vec` 和 `Result`）都包含在**标准前奏**里，会被自动导入。Rust 就好像每个模块（包括根模块），都会以下面的导入声明开始：

```
use std::prelude::v1::*;
```

这个标准的前置模块（prelude）中包含了几十个常用的特型和类型。但**不**包含 `std`。因此如果模块引用 `std`，则必须明确导入它，就像下面这样：

```
use std;
```

一般来说，导入正在使用的 std 的特定功能会更有意义。

第 2 章曾提到过，库有时候会提供名为 prelude 的模块。但 std::prelude::v1 是唯一会自动导入的前置模块。将一个模块命名为 prelude 只不过是一个约定，意在告诉用户这个模块应该使用 * 导入。

8.2.4　特性项，Rust的基础

模块由**特性项**构成。而特性项也分很多种，下面列出的只是这门语言中主要的特性：

- **函数**

  我们已经看到很多了。

- **类型**

  用户定义类型通过 struct、enum 和 trait 关键字定义。本书会为它们各自开辟一章的篇幅，大家很快就能看到。一个简单的结构体看起来是这样的：

```
pub struct Fern {
    pub roots: RootSet,
    pub stems: StemSet
}
```

  结构体的字段，即便是私有字段，也可以在声明结构体的模块中的任意位置访问到。在模块外部，则只有公有字段可以访问到。

  事实证明，通过模块进行访问控制而不是像 Java 或 C++ 中通过类来实现，对软件设计而言出人意料地有用。它不仅可以避免繁复的“getter”和“setter”方法，很大程度上还消除了类似 C++ 中 friend 这样的声明。一个模块可以定义多个紧密协作的类型，比如 frond::LeafMap 和 frond::LeafMapIter，它们可以随时相互访问各自的私有字段，同时又可以对程序的其他部分隐藏相应的实现细节。

- **类型别名**

  前面看到过，type 关键字可以像 C++ 中的 typedef 那样使用，为现有类型声明一个新名字：

```
type Table = HashMap<String, Vec<String>>;
```

  这里声明的类型 Table 就是这个特定的 HashMap 的简写。

```
fn show(table: &Table) {
    ...
}
```

- **impl 块**

  方法通过 impl 块添加到类型上：

```
impl Cell {
    pub fn distance_from_origin(&self) -> f64 {
        f64::hypot(self.x, self.y)
    }
}
```

这个语法将在第 9 章进行解释。不能将 impl 块标记为 pub，要标记，必须标记个别的方法，以便让它们在当前模块外部可见。

私有方法，类似私有的结构体字段，在声明它们的模块的任何位置都是可见的。

- **常量**

  const 关键字定义常量。这个语法与 let 的区别在于，它可以标记为 pub，而且必须写明类型。同样，像 UPPERCASE_NAMES 这样全部使用大写字母也是常量命名的惯例：

```
pub const ROOM_TEMPERATURE: f64 = 20.0; // 摄氏度
```

  static 关键字定义静态特性项，跟常量差不太多：

```
pub static ROOM_TEMPERATURE: f64 = 68.0; // 华氏度
```

常量有点类似 C++ 的 #define，即它的值会编译到代码中使用它的每个地方。静态变量则是在程序运行前就已经存在且会持续存在直到程序退出的值。常量在代码中通常用于保存魔法数值和字符串。静态变量则用于保存大量数据，或者用于借用对常量值的引用。

没有可修改（mut）的常量。静态变量可以标记为 mut，但正如第 5 章所讨论的，Rust 没办法保证对可修改静态变量的专有访问权。因此，可修改静态变量本质上不是线程安全的，绝不能在安全代码中使用：

```
static mut PACKETS_SERVED: usize = 0;

println!("{} served", PACKETS_SERVED); // 错误：使用可修改静态变量
```

Rust 不提倡使用全局可修改状态。至于用什么替代，参见 19.3.11 节关于“全局变量”的讨论。

- **模块**

  关于模块已经说了不少了。如前所见，模块可以包含子模块，而且跟其他有名字的特性项一样，模块可以是公有的也可以是私有的。

- **导入**

  use 和 extern crate 声明也是特性项。即使它们只是别名，也可以是公有的：

```
// 在plant_structures/mod.rs中
...
pub use self::leaves::Leaf;
pub use self::roots::Root;
```

  这意味着 Leaf 和 Root 是 plant_structures 模块的公有特性项。同时，它们还是 plant_structures::leaves::Leaf 和 plant_structrues::roots::Root 的简化别名。

  标准前置模块就是像这样写成了一系列的 pub 导入来定义的。

- **extern 块**

  extern 块用于声明用其他语言（通常是 C 或 C++）编写的函数集合，以便 Rust 代码可以调用它们。第 21 章将介绍 extern 块。

对于声明了却没有使用的特性项，Rust 会给出警告：

```
warning: function is never used: `is_square`
  --> src/crates_unused_items.rs:23:9
   |
23 | /         pub fn is_square(root: &Root) -> bool {
24 | |             root.cross_section_shape().is_square()
25 | |         }
   | |_________^
   |
```

这里的警告可能会令人费解，因为存在两种非常不一样的可能。一是这个函数在此时此刻的确是死代码；二是你本来打算在其他包里使用它。如果是后一种可能，则需要将这个函数乃至**整个包含模块**都标记为公有。

8.3　将程序作为库发布

随着蕨类植物模拟程序的开工，你会发现自己需要更多程序。假设现在已经有了一个命令行程序，其可以运行模拟并把结果保存到一个文件中。现在，你想再写几个程序，用于对保存在文件中的结果进行科学分析、以 3D 形式实时显示植物生成过程，以及渲染超写实的照片，等等。所有这些程序都需要共享基本的蕨类植物模拟代码。因此，你需要把它改成一个库。

第一步是要把已有项目分成两部分：一个要成为库的包，包含所有共享代码；一个可执行文件，包含仅供已有的命令行程序使用的代码。

为示范如何做，让我们使用下面这个非常简单的示例程序：

```
struct Fern {
    size: f64,
    growth_rate: f64
}

impl Fern {
    /// 模拟蕨类植物一天的生长
    fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// 运行并模拟指定天数的生长状况
fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}

fn main () {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
```

```
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}
```

假设这个程序有一个非常简单的 Cargo.toml 文件：

```
[package]
name = "fern_sim"
version = "0.1.0"
authors = ["You <you@example.com>"]
```

把这个程序作为库发布很简单，步骤如下。

1. 将 src/main.rs 重命名为 src/lib.rs。
2. 给 src/lib.rs 中要成为库的公有特性的项添加 `pub` 关键字。
3. 把 main 函数临时转移到其他地方（稍后还会用到它）。

此时的 src/lib.rs 文件变成了这样：

```
pub struct Fern {
    size: f64,
    growth_rate: f64
}

impl Fern {
    /// 模拟蕨类植物一天的生长
    pub fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// 运行并模拟指定天数的生长状况
pub fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}
```

注意，Cargo.toml 文件未作任何修改。这是因为最小化的 Cargo.toml 文件可以让 Cargo 按照默认配置行事。默认情况下，`cargo build` 会从源代码目录中查找文件，然后决定如何构建。看到了 src/lib.rs，它就知道要构建的是一个库。

src/lib.rs 中的代码构成了库的**根模块**。使用这个库的其他包只能访问这个根模块中的公有特性。

8.4 src/bin目录

让原来的命令行程序 `fern_sim` 再运行起来也很简单，Cargo 内置支持与作为库的代码放在一起的小程序。

事实上，Cargo 本身就是以这种方式写的。它的大部分代码写在一个 Rust 库里。而本书中随处可见的命令行程序 `cargo` 只是一个小小的包装程序，所有重量级工作实际上都会交给

这个库来做。这个库和命令行程序都位于同一个代码仓库中。

我们也可以将程序和库放在同一份代码中。把以下代码放到一个名为 src/bin/efern.rs 的文件中：

```
extern crate fern_sim;
use fern_sim::{Fern, run_simulation};

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}
```

这个 `main` 函数就是刚才暂时挪走的那个。在它的前面，又加上了 `extern crate` 声明，因为这个程序要使用 `fern_sim` 库作为外部包。然后，从这个库中导入了 `Fern` 和 `run_simulation`。

因为这个文件在 src/bin 目录下，所以 Cargo 会在下次运行 `cargo build` 时同时编译 `fern_sim` 库和这个程序。然后就可以使用 `cargo run --bin efern` 来运行 `efern` 程序了。下面是运行它的输出，使用 `--verbose` 可以看到 Cargo 正在运行的命令：

```
$ cargo build --verbose
   Compiling fern_sim v0.1.0 (file:///.../fern_sim)
     Running `rustc src/lib.rs --crate-name fern_sim --crate-type lib ...`
     Running `rustc src/bin/efern.rs --crate-name efern --crate-type bin ...`
$ cargo run --bin efern --verbose
       Fresh fern_sim v0.1.0 (file:///.../fern_sim)
     Running `target/debug/efern`
final fern size: 2.7169239322355985
```

同样，仍然不需要对 Cargo.toml 做任何修改，因为 Cargo 的默认配置就是查看源文件，然后把事情搞定。Cargo 会自动将 src/bin 中的 .rs 文件作为要构建的额外程序。

当然，现在 `fern_sim` 是一个库，因此还有另一种选择。可以把这个程序写成一个独立的项目，保存到一个完全分开的目录中，然后在它的 Cargo.toml 中将 `fern_sim` 作为依赖加上：

```
[dependencies]
fern_sim = { path = "../fern_sim" }
```

或许这就是你对即将开发的其他蕨类植物模拟程序要做的。而 src/bin 目录只适合 `efern` 这样简单的程序。

8.5 属性

Rust 程序中的任何特性项都可以用**属性**（attribute）来修饰。属性是 Rust 中写给编译器看的各种指令和建议的普适语法。比如，假设你收到了如下警告：

```
libgit2.rs: warning: type `git_revspec` should have a camel case name
    such as `GitRevspec`, #[warn(non_camel_case_types)] on by default
```

但是，你这样命名是有原因的，所以希望 Rust 在碰到这个名字时“闭嘴”。那么在相应类型上添加一个 #[allow] 属性，就可以禁用上面的警告：

```
#[allow(non_camel_case_types)]
pub struct git_revspec {
    ...
}
```

条件编译作为一个特性也是使用属性 #[cfg] 写出来的：

```
// 只在针对安卓编译时包含此模块
#[cfg(target_os = "android")]
mod mobile;
```

#[cfg] 的全部语法可以查看 Rust 参考，表 8-2 列出了最常用的一部分。

表8-2：常用的#[cfg]语法

#[cfg(...)]选项	何时启用
test	启用测试（以 cargo test 或 rustc --test 编译时）
debug_assertions	启用调试断言（通常用于非优化构建）
unix	为 Unix（包括 macOS）编译
windows	为 Windows 编译
target_pointer_width = "64"	针对 64 位平台。另一个可能的值是 "32"
target_arch = "x86_64"	针对 x86-64 架构。其他值还有："x86"、"arm"、"aarch64"、"powerpc"、"powerpc64" 和 "mips"
target_os = "macos"	为 macOS 编译。其他值还有："windows"、"ios"、"android"、"linux"、"openbsd"、"netbsd"、"dragonfly" 和 "bitrig"
feature = "robots"	启用用户定义的名为 "robots" 的特性（以 cargo build --feature robots 或 rustc --cfg feature='"robots"' 编译时）。特性在 Cargo.toml 的 [features] 部分声明
not(*A*)	*A* 不满足时要提供一个函数的两个不同实现，将其中一个标记为 #[cfg(X)]，另一个标记为 #[cfg(not(x))]
all(*A*,*B*)	*A* 和 *B* 都满足时（等于 &&）
any(*A*,*B*)	*A* 或 *B* 满足时（等于 ||）

偶尔，可能需要对函数的行内扩展（通常会交给编译器去做的一项优化）进行一些微观控制。此时可以使用 #[inline] 属性：

```
/// 由于相互间有渗透，调整两个相邻细胞间的离子级别
#[inline]
fn do_osmosis(c1: &mut Cell, c2: &mut Cell) {
    ...
}
```

有一种情况是，没有 #[inline] 行内扩展就**不会**发生。如果函数或方法在一个包里定义，但在另一个包里调用，那么 Rust 就不会将其在行内扩展，除非它是泛型的（有类型参数）或者明确标记为 #[inline]。

否则，编译器会将 #[inline] 看成一个建议。Rust 也支持更激进的 #[inline(always)]，要

求每处调用都将函数进行行内扩展以及 #[inline(never)]，要求永远不要行内化。

有些属性，比如 #[cfg] 和 #[allow]，可以加到整个模块中并应用于其中所有的特性。而另外一些属性，比如 #[test] 和 #[inline]，则只能添加到个别特性项。作为一种普适语法，每个属性都是定制的，都有自己支持的参数。Rust 参考文件详细介绍了支持的属性。

要将属性添加给整个包，需要在 main.rs 或 lib.rs 文件的顶部、任何特性项之前添加，而且要写 #! 而非 #。比如：

```
// libgit2_sys/lib.rs
#![allow(non_camel_case_types)]

pub struct git_revspec {
    ...
}

pub struct git_error {
    ...
}
```

这里的 #! 告诉 Rust 将属性添加给整个特性项，而不是其后面的个别特性项。对这个例子而言，#![allow] 属性会添加到整个 libgit2_sys 包，而不仅仅是 struct git_revspec。

#! 也可以出现在函数、结构体等的内部，但通常只出现在文件开头，用于给整个模块或包添加属性。有些属性始终要使用 #! 语法，因为只能将它们添加给整个包。

比如，#![feature] 属性用于开启 Rust 语言和库的**不安全**特性（或者说实验性特性——因此可能会有 bug 或者将来有可能会被更改或删除）。举个例子，写作本书时，Rust 有一个实验性特性，其支持 128 位整数类型 i128 和 u128。但由于这些类型是实验性的，因此要使用的话，只能 (1) 安装 Rust 的每夜构建版且 (2) 明确声明你的包要使用它们：

```
#![feature(i128_type)]

fn main() {
    // 做我的数学作业，Rust!
    println!("{}", 9204093811595833589_u128 * 19973810893143440503_u128);
}
```

随着时间推移，Rust 团队会在某一天将某个实验性特性列为**稳定化**，之后这个特性就会成为语言标准的一部分。那时候，#![feature] 属性就变成多余的了，Rust 会生成一个警告来建议你删除它。

8.6 测试和文档

正如 2.3 节所说，Rust 内置了一个简单的单元测试框架。测试就是以 #[test] 属性标记的普通函数。

```
#[test]
fn math_works() {
    let x: i32 = 1;
    assert!(x.is_positive());
```

```
        assert_eq!(x + 1, 2);
    }
```

cargo test 会运行项目中的所有测试。

```
$ cargo test
   Compiling math_test v0.1.0 (file:///.../math_test)
     Running target/release/math_test-e31ed91ae51ebf22

running 1 test
test math_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

（稍后还会讨论关于“文档测试”的内容。）

无论你的包是一个可执行文件还是一个库，结果都是一样的。要想运行特定的测试，可以给 Cargo 传入相应的参数，比如 cargo test match 会运行其名字中包含 match 的所有测试。

测试中经常会用到 Rust 标准库中的两个宏：assert! 和 assert_eq!。assert!(expr) 在 expr 为 true 时成功；否则，会诧异并导致测试失败。assert_eq!(v1, v2) 其实跟 assert!(v1 == v2) 一样，区别仅在于如果断言失败，那么错误消息会显示两个值。

在一般的代码里可以使用这两个宏来检查不变性（invariant），但要注意即使在发布构建中也会包含 assert! 和 assert_eq!。可以使用 debug_assert! 和 debug_assert_eq!，而不是编写只在调试构建中检查的断言：

要测试错误用例，可以给测试添加 #[should_panic] 属性：

```
/// 正如上一章声明过的那样，这个测试只在被零除导致诧异时才会通过
#[test]
#[should_panic(expected="divide by zero")]
fn test_divide_by_zero_error() {
    1 / 0;  // 应该诧异!
}
```

标记为 #[test] 的函数会被有条件地编译。如果运行的是 cargo test，Cargo 构建的结果中就会包含程序、测试并启用测试套件。而简单的 cargo build 或 cargo build --release 会跳过测试代码。这意味着单元测试代码可以与它测试的代码待在一起，根据需要访问内部实现细节，同时又没有运行时消耗。不过，有可能会导致某些警告。比如：

```
fn roughly_equal(a: f64, b: f64) -> bool {
    (a - b).abs() < 1e-6
}

#[test]
fn trig_works() {
    use std::f64::consts::PI;
    assert!(roughly_equal(PI.sin(), 0.0));
}
```

在测试构建中，一切如常。而在非测试构建中，roughly_equal 是用不到的，因此 Rust 会给出警告：

```
$ cargo build
   Compiling math_test v0.1.0 (file:///.../math_test)
warning: function is never used: `roughly_equal`
 --> src/crates_unused_testing_function.rs:7:1
  |
7 | / fn roughly_equal(a: f64, b: f64) -> bool {
8 | |     (a - b).abs() < 1e-6
9 | | }
  | |_^
  |
   = note: #[warn(dead_code)] on by default
```

所以，实践中当测试多到需要支持代码时，通行的做法是把它们放到一个 `tests` 模块中，并使用 `#[cfg]` 属性将整个模块声明为仅供测试使用：

```
#[cfg(test)] // 只在测试构建时包含此模块
mod tests {
    fn roughly_equal(a: f64, b: f64) -> bool {
        (a - b).abs() < 1e-6
    }

    #[test]
    fn trig_works() {
        use std::f64::consts::PI;
        assert!(roughly_equal(PI.sin(), 0.0));
    }
}
```

Rust 的测试套件会通过多个线程同时运行多个测试，这也是 Rust 代码默认就能保证线程安全的一个小小惊喜。（如果想禁用多线程，要么每次只运行一个测试，如 `cargo test testname`，要么把 `RUST_TEST_THREADS` 环境变量设置为 1。）这也就意味着，从技术角度讲，第 2 章的曼德布洛特程序并非该章第二个多线程程序，而是第三个！第一个是 2.3 节运行的 `cargo test`。

8.6.1 集成测试

蕨类植物模拟程序的代码还在不断增长。我们已经决定把所有核心功能都转移到一个库里，以便多个可执行文件使用。如果能像外部用户一样，把 fern_sim.rlib 当成一个外部包，从而对这个库进行各种测试就好了。另外，还有一些测试需要加载保存的二进制模拟结果，但把那些大型测试文件放在 src 目录里显得太笨重了。集成测试能帮忙解决这两个问题。

集成测试是放在 tests 目录中的 .rs 文件，tests 目录与项目的 src 目录放在一起。运行 `cargo test` 时，Cargo 会把每个集成测试都编译成一个独立的包，并将其链接到你的库和 Rust 测试套件。下面是一个例子：

```
// tests/unfurl.rs - Fiddleheads unfurl in sunlight

extern crate fern_sim;
use fern_sim::Terrarium;
use std::time::Duration;
```

```
#[test]
fn test_fiddlehead_unfurling() {
    let mut world = Terrarium::load("tests/unfurl_files/fiddlehead.tm");
    assert!(world.fern(0).is_furled());
    let one_hour = Duration::from_secs(60 * 60);
    world.apply_sunlight(one_hour);
    assert!(world.fern(0).is_fully_unfurled());
}
```

注意，集成测试的开头包含了 `extern crate` 声明，因为它把 `fern_sim` 当成了一个外部库来用。集成测试的关键在于它以外部包的眼光来看待你的代码（就像用户一样），测试你的公共 API。

`cargo test` 既运行单元测试也运行集成测试。如果只想运行特定文件（比如 tests/unfurl.rs）中的集成测试，可以使用命令 `cargo test --test unfurl`。

8.6.2 文档

`cargo doc` 命令可以为你的库创建 HTML 文档：

```
$ cargo doc --no-deps --open
 Documenting fern_sim v0.1.0 (file:///.../fern_sim)
```

其中的 `--no-deps` 选项告诉 Cargo 只为 `fern_sim` 自身生成文档，不考虑它依赖的包。而 `--open` 选项告诉 Cargo 生成文档后就在浏览器中打开。

图 8-2 展示了生成的 HTML 文档。Cargo 把新的文档文件保存到 target/doc 目录中。文档首页是 target/doc/fern_sim/index.html。

Click or press 'S' to search, '?' for more options...

Crate fern_sim **[−] [src]**

[−] Simulate the growth of ferns, from the level of individual cells on up.

Reexports

```
pub use plant_structures::Fern;
pub use simulation::Terrarium;
```

Modules

cells	The simulation of biological cells, which is as low-level as we go.
plant_structures	Higher-level biological structures.
simulation	Overall simulation control.
spores	Fern reproduction.

图 8-2：`rustdoc` 生成的文档示例

这些文档基于库中的 pub 特性以及它们对应的**文档注释**生成。本章已经展示了几个文档注释了，比如：

```
/// 模拟通过成熟分裂产生孢子
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}
```

不过，Rust 在看到由 3 个斜杠开头的注释时，会将其作为一个 #[doc] 属性来看待。换句话说，在 Rust 看来，前面的代码跟下面这段代码是一样的：

```
#[doc = "模拟通过成熟分裂产生孢子"]
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}
```

在编译或测试库时，这些属性会被忽略。而在生成文档时，公共特性的文档注释就会包含在输出中。

类似地，以 //! 开头的注释会被看成 #![doc] 属性，从而添加到相应的包含特性，通常是模块或包中。比如，fern_sim/src/lib.rs 文件可能会以下面的注释开头：

```
//! 模拟蕨类植物生长
//! 从单个细胞开始
```

文档注释的内容会被按照 Markdown（一种简化的 HTML 格式）来解析。星号用于表示 *斜体* 和 **粗体**，空行则被视为另起一段，等等。不过，直接写 HTML 也是可以的。文档注释中的任何 HTML 标签也都原样不变地复制到文档中。

文本中的代码可以用反引号 `code` 标记。在输出中，反引号标记的部分会以等宽代码体展示。代码段需要缩进 4 个空格：

```
/// 文档注释中的代码块：
///
///     if everything().works() {
///         println!("ok");
///     }
```

你也可以使用 Markdown 分隔的代码块，最终效果相同。

````
/// 另一个相同的代码块，只是写法不同：
///
/// ```
/// if everything().works() {
///     println!("ok");
/// }
/// ```
````

不管代码块怎么写，最有意思的是 Rust 会将文档注释中的代码块自动转换为测试。

8.6.3 文档测试

在运行 Rust 库中的测试时，Rust 会检查所有出现在文档中的代码是否可以运行以及是否

有效。为此，Rust 会提取出注释中的每个代码块，将其作为独立的可执行包进行编译，再将它链接到你的库，最后运行。

下面是一个独立的文档测试（doc-test）的例子。运行 `cargo new ranges` 创建一个新项目，把以下代码放到 ranges/src/lib.rs 中：

```
use std::ops::Range;

/// 如果两个范围重合，就返回true
///
///     assert_eq!(ranges::overlap(0..7, 3..10), true);
///     assert_eq!(ranges::overlap(1..5, 101..105), false);
///
/// 如果有一个范围为空，则不认为是重合
///
///     assert_eq!(ranges::overlap(0..0, 0..10), false);
///
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool {
  r1.start < r1.end && r2.start < r2.end &&
      r1.start < r2.end && r2.start < r1.end
}
```

文档注释中的这两小段代码出现在了 `cargo doc` 生成的文档中，如图 8-3 所示。

Function ranges::overlap [−] [src]

```
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool
```

[−] Return true if two ranges overlap.

```
assert_eq!(ranges::overlap(0..7, 3..10), true);
assert_eq!(ranges::overlap(1..5, 101..105), false);
```

If either range is empty, they don't count as overlapping.

```
assert_eq!(ranges::overlap(0..0, 0..10), false);
```

图 8-3：文档中出现了文档测试

这两段代码也变成了两个单独的测试：

```
$ cargo test
   Compiling ranges v0.1.0 (file:///.../ranges)
...
   Doc-tests ranges

running 2 tests
test overlap_0 ... ok
```

```
test overlap_1 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

如果给 Cargo 传入 `--verbose` 标记，就会发现它是通过 `rustdoc --test` 来运行这两个测试的。Rustdoc 会把每个代码示例都保存到一个独立的文件中，并给它们加上相应的模板代码，从而得到两个程序。下面是第一个：

```
extern crate ranges;
fn main() {
    assert_eq!(ranges::overlap(0..7, 3..10), true);
    assert_eq!(ranges::overlap(1..5, 101..105), false);
}
```

这是第二个：

```
extern crate ranges;
fn main() {
    assert_eq!(ranges::overlap(0..0, 0..10), false);
}
```

如果这两个程序都能编译并运行成功，测试就会通过。

这两个代码示例中都包含断言，不过这只是因为在这种情况下，断言可以让文档看起来更好理解。文档测试并不鼓励把所有测试都放到注释中。相反，开发者尽力写出最好的文档，而 Rust 会帮你确保文档中的代码示例能够实际地编译和运行。

有时候，就算很简单的一行代码调用都会涉及很多细节。比如，要依赖之前导入或编写的代码，而且这也是代码能编译的必要条件，只是没有必要在文档中展示它们罢了。要隐藏代码示例中的某些行，可以在这些行前面加上 `#`，然后再加一个空格：

```
/// 让阳光照过来，并按指定时间来运行模拟程序
///
///     # use fern_sim::Terrarium;
///     # use std::time::Duration;
///     # let mut tm = Terraium::new();
///     tm.apple_sunlight(Duration::from_secs(60));
pub fn apply_sunlight(&mut self, time: Duration) {
    ...
}
```

有时候在文档中展示完整的示例代码（包括 `main` 函数和 `extern crate` 声明）是很有必要的。显然，如果这些代码出现在了文档示例中，那就不需要 Rustdoc 再自动补全它们了。否则结果将无法编译。为此，Rustdoc 会将包含字符串 `fn main` 的代码块当作完整的程序，不给它添加任何代码。

特定的代码块可以禁用测试。要告诉 Rust 可以编译示例，但并不实际运行它，可以使用代码块定界符号并加上 `no_run` 注解：

````
/// 将本地结果上传到在线相册
///
/// ```no_run
/// let mut session = fern_sim::connect();
````

```
/// session.upload_all();
/// ```
pub fn upload_all(&mut self) {
    ...
}
```

如果代码不希望示例被编译，那就把 `no_run` 替换成 `ignore`。如果代码块根本不是 Rust 代码，则需要使用语言的名字，比如 `c++` 或 `sh`，纯文本就直接用 `text`。`rustdoc` 并不认得几百种编程语言，它只是把自己不认得的注解当成表示不是 Rust 代码块。这样也会禁用文档测试中的代码高亮。

8.7 指定依赖

我们已经知道了一种告诉 Cargo 到哪里去寻找项目依赖包源代码的方式：版本号。

```
image = "0.6.1"
```

除此之外，还有几种指定依赖的方式，以及关于使用哪个版本还有一些细节要交代，因此本节就专门讨论一下。

首先，你可能想要使用没有发布到 crates.io 上的依赖。为此，可以指定 Git 仓库的地址和修订版本：

```
image = { git = "https://github.com/Piston/image.git", rev = "528f19c" }
```

这里这个包是开源的，托管在 GitHub 上。其实要指定托管在你们公司网络上的私有 Git 仓库地址，也是一样的。如以上代码所示，可以指定要使用哪个 `rev`、`tag` 或 `branch`。（这是告诉 Git 要获取源代码的哪次提交的所有方式。）

另一种方式是指定包含依赖包源代码的目录：

```
image = { path = "vendor/image" }
```

如果你们团队只使用一个版本控制仓库，其中包含几个依赖包的源代码，或者包含完整的依赖图，那这种方式比较方便。每个包都可以使用相对路径来指定自己的依赖。

能够对依赖控制到这么细粒度是很有用的。假如你发现某个开源包不太合用，那可以先复制一份：只要在 GitHub 上点一下“Fork”，然后修改 Cargo.toml 文件中一行代码即可。接下来的 `cargo build` 会立即切换到你复制的版本，而不再使用官方的版本。

8.7.1 版本

对于在 Cargo.toml 中写的 `image = "0.6.1"`，Cargo 的解释并没有那么严格。它会使用与 0.6.1 版兼容的最新版本的 `image`。

兼容性的判断基本上遵循“语义化版本”的思想。

- 以 0.0 开头的版本过于原始，Cargo 不会假设它与任何其他版本兼容。
- 以 0.*x*（其中 *x* 不是 0）开头的版本，会被认为同其他以 0.*x* 开头的版本兼容。比如前面我们指定的是 `image` 的 0.6.1 版，而 Cargo 有可能会使用 0.6.3 版（如果有的话）。（这并

不是“语义化版本”对 0.*x* 的标准解读，但这个规则实在太有用了，无法割舍。）

- 如果项目达到 1.0 版,则只有新的主版本才会破坏兼容性。因此如果你指定的是 2.0.1 版，那么 Cargo 有可能会使用 2.17.99 版，但绝不会是 3.0 版。

版本编号默认是灵活的，否则选择版本很快就会成为大问题。假设有一个库 `libA`，它使用了 `num = "0.1.31"`，而另一个库 `libB` 使用的是 `num = "0.1.29"`。如果要求版本号严格匹配，那将不会有项目可以同时使用这两个库。因此允许 Cargo 使用任何兼容的版本是更加实际的默认选择。

同样，不同项目对依赖和版本也有不同需求。因此，指定版本时可以使用操作符，其既可以指定严格匹配，也可以指定版本范围，如表 8-3 所示。

表8-3：使用操作符指定版本

Cargo.toml中的写法	含　义
`image = "=0.10.0"`	只使用 0.10.0 版
`image = ">=1.0.5"`	使用 1.0.5 版或更高版本（甚至 2.9 版都可以，如果有的话）
`image = ">1.0.5 <1.1.9"`	使用大于 1.0.5 但小于 1.1.9 的版本
`image = "<=2.7.10"`	使用小于或等于 2.7.10 的版本

另一种指定版本的策略是使用通配符 `*`，但并不多见。这是告诉 Cargo 任何版本都可以。除非另外一些 Cargo.toml 文件里有更具体的限制，否则 Cargo 会使用最新版本。要了解更详细的版本控制方式，可以参考 doc.crates.io 上的 Cargo 文档。

注意，版本兼容性规则意味着不能纯粹为了市场推广而任意选择版本号。版本号真的有它的实际用途：它是包的维护者与使用者的一种约定。如果你维护的一个包的版本是 1.7，在删除了一个函数或者做了其他不能向后兼容的改动后，则必须要将版本号跳到 2.0。如果你非要称它为 1.8，就意味着新版本仍然兼容 1.7，结果用户可能会无法构建代码。

8.7.2　Cargo.lock

Cargo.toml 中的版本号是有意这么灵活处理的，但我们并不想让 Cargo 每次构建都升级一次依赖版本。想象一下，我们正在紧张地调试代码，而 `cargo build` 运行后升级到了一个新版本的依赖。有时候这是具有破坏性的。调试过程中的任何变更都应该避免。事实上，说到依赖的库，它的版本始终不变才是最好的。

Cargo 为此提供了内置机制以避免上述情况发生。在第一次构建项目时，Cargo 会输出一个 Cargo.lock 文件，记录它使用的每个包的确切版本号。后续构建都会参考这个文件，并继续使用相同的版本。Cargo 只会在你告诉它升级时才升级，比如你手工修改了 Cargo.toml 文件中的版本号，或者运行了 `cargo update`：

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating libc v0.2.7 -> v0.2.11
    Updating png v0.4.2 -> v0.4.3
```

`cargo update` 只会升级到与 Cargo.toml 中指定版本兼容的最新版本。如果你指定的是 `image = "0.6.1"`，而你想升级到 0.10.0，那必须在 Cargo.toml 中修改。下一次构建时，Cargo 就会把 `image` 库升级到这个新版本，并把新版本号保存到 Cargo.lock 里。

前面的例子展示了 Cargo 更新了两个包，它们都托管在 crates.io 上。对于保存在 Git 代码库的依赖，情况也是类似的。假设 Cargo.toml 文件中有如下依赖：

```
image = { git = "https://github.com/Piston/image.git", branch = "master" }
```

如果存在 Cargo.lock 文件，则 `cargo build` 不会从 Git 仓库中拉取新的变更。在读取 Cargo.lock 后，Cargo 仍然会沿用上一次使用的修订版本。但是，`cargo update` 会重新拉取 `master` 分支，因此下一次构建就会使用该分支最新提交的代码。

Cargo.lock 是自动生成的，通常不建议手工修改。但是，如果你的项目是一个可执行文件，那应该把 Cargo.lock 提交到版本控制系统。这样，任何构建你项目的人都会拿到相同的版本。Cargo.lock 文件的历史将记录依赖更新的过程。

如果你的项目是一个普通的 Rust 库，那就不用提交 Cargo.lock 了，因为这个库的下游用户会生成包含自己完整依赖图版本信息的 Cargo.lock 文件，他们会忽略库里的 Cargo.lock 文件。如果恰巧你的项目是一个共享库（例如输出是 .dll、.dylib 或 .so 文件），没有这种下游的 `cargo` 用户，那就应该提供 Cargo.lock。

Cargo.toml 灵活的版本机制使在项目中使用 Rust 库变得非常方便，同时也能最大化利用库的兼容性。Cargo.lock 的记录可以确保在不同机器上得到一致、可再现的构建。将这两个文件合在一起，很大程度上能够帮开发者避免依赖管理的问题。

8.8 把包发布到crates.io

你已经决定把你的蕨类植物模拟库作为开源软件发布。祝贺你！这一步很简单。

首先，让 Cargo 帮你打个包：

```
$ cargo package
warning: manifest has no description, license, license-file, documentation,
homepage or repository. See http://doc.crates.io/manifest.html#package-metadata
for more info.
   Packaging fern_sim v0.1.0 (file:///.../fern_sim)
   Verifying fern_sim v0.1.0 (file:///.../fern_sim)
   Compiling fern_sim v0.1.0 (file:///.../fern_sim/target/package/fern_sim-0.1.0)
```

`cargo package` 命令会创建一个文件（这里是 target/package/fern_sim-0.1.0.crate），其中包含库的所有源文件，以及 Cargo.toml。这个文件就是你要上传到 crates.io 与世界共享的（可以通过 `cargo package --list` 来查看其中包含什么文件）。Cargo 随后会基于这个 .crate 文件构建库，就像最终用户一样，以确保没有问题。

在前面可以看到，Cargo 认为 Cargo.toml 中缺少一些对下游用户来说非常重要的信息，比如你的代码以什么许可来发布。警告信息中的 URL 是很好的参考资源，因此这里就不详细解释了。简单来说，只要在 Cargo.toml 中添加以下几行内容就可以消除警告：

```
[package]
name = "fern_sim"
version = "0.1.0"
authors = ["You <you@example.com>"]
license = "MIT"
homepage = "https://fernsim.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsim.example.com/docs"
description = """
Fern simulation, from the cellular level up.
"""
```



在 crates.io 上发布了自己的包以后，任何下载它的人都可以看到这个 Cargo.toml 文件。因此，如果 `authors` 字段中包含你不想对外公布的电子邮件地址，那现在就可以把它改掉。

此时还应该再考虑另一个问题，即你的 Cargo.toml 文件中可能使用了 `path` 指定其他包的位置，参见 8.7 节的“指定依赖”：

```
image = { path = "vendor/image" }
```

对你和你的团队来说，这样可能没问题。但可想而知，如果其他人下载了这个 `fern_sim` 库，那他们电脑里的文件和目录不可能跟你们一样。因此 Cargo 此时会**忽略**自动下载的库中的 `path` 字段，而这可能导致构建错误。不过解决办法倒也简单：如果你准备把库发布到 crates.io，那它的依赖也应该在 crates.io 上。此时应该指定版本号而不是使用 `path`。

```
image = "0.6.1"
```

如果你愿意，可以既指定 `path`（本地构建时优先使用），也指定 `version`（以便其他用户使用）：

```
image = { path = "vendor/image", version = "0.6.1" }
```

当然，在这种情况下，必须确保二者是同步的。

最后，在发布包之前，必须先登录到 crates.io 并取得 API 密钥。这一步很简单，只要有 crates.io 的账号，在“账号设置”里，就可以看到一个 `cargo login` 命令，类似下面这个：

```
$ cargo login 5j0dV54BjlXBpUUbfIj7G9DvNl1vsWW1
```

Cargo 会把这个密钥保存到一个配置文件中，注意这个 API 密钥就像密码一样要保密。所以要记住只在你自己控制的计算机中运行这个命令。

通过密钥登录后，最后一步是运行 `cargo publish`：

```
$ cargo publish
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Uploading fern_sim v0.1.0 (file:///.../fern_sim)
```

看到这个消息，就说明你的库已经加入了 crates.io 上面成千上万的开源库的行列。

8.9　工作空间

随着项目规模的增大，最终你可能会写出很多个包来。这些包并列存在于一个源代码仓库中：

```
fernsoft/
├── .git/...
├── fern_sim/
│   ├── Cargo.toml
│   ├── Cargo.lock
│   ├── src/...
│   └── target/...
├── fern_img/
│   ├── Cargo.toml
│   ├── Cargo.lock
│   ├── src/...
│   └── target/...
└── fern_video/
    ├── Cargo.toml
    ├── Cargo.lock
    ├── src/...
    └── target/...
```

Cargo 会保证每个包都有自己的构建目录：target，其中包含一份这个包所有依赖的独立构建。这些构建目录完全是独立的。即便两个包有相同的依赖，也不能共享任何编译后的代码。这显然有点浪费。

使用 Cargo **工作空间**（workspace）可以节省编译时间和磁盘空间。所谓工作空间，就是共享相同构建目录和 Cargo.lock 文件的一组包。

要使用工作空间，只需在存储库的根目录下创建一个 Cargo.toml 文件，并把下面几行放进去：

```
[workspace]
members = ["fern_sim", "fern_img", "fern_video"]
```

其中 `fern_sim` 等是包含各自包的子目录的名称。然后，把这些子目录中剩下的 Cargo.lock 文件和 target 目录都删除。

做完这些之后，无论在任何包中运行 `cargo build`，都会自动在根目录中创建一个共享的构建目录（这里就是 fernsoft/target），所有包共享。运行命令 `cargo build --all` 会构建当前工作空间中所有的包。同样，`cargo test` 和 `cargo doc` 也都接收 `--all` 选项。

8.10　还有惊喜

如果这些还不足以让你高兴，Rust 社区还为你提供了一些好东西。

- 当你在 crates.io 上发布了自己的开源包之后，你的文档会自动发布到 docs.rs（感谢 Onur Aslan）。

- 如果你的项目在 GitHub 上，Travis CI 可以在你每次推送代码时自动构建和测试。设置也极其简单，详细信息请参考 travis-ci.org。如果你已经熟悉 Travis，那可以从这个 .travis.yml 开始。

```
language: rust
rust:
  - stable
```

- 可以基于包的顶级文档注释生成一个 README.md 文件。这个功能是由 Livio Ribeiro 开发的一个第三方 Cargo 插件提供的。运行 `cargo install readme` 即可安装这个插件，然后可以通过 `cargo readme --help` 来学习如何使用它。

当然，令人兴奋的东西还不止这些。

Rust 是一门新语言，但从一开始设计它就是要支持大型、重量级的项目。迄今为止，Rust 活跃的社区已经涌现了一大批优秀的工具。系统程序员们也有很酷的东西可以玩了。

第 9 章

结构体

> 很久以前，牧羊人就是通过观察外在特征是否满足同构条件来确定两群羊是否属于同一类的。
>
> ——John C. Baez，James Dolan，“Categorification”

Rust 的结构体，有时候也叫**结构**，类似于 C 和 C++ 中的 `struct` 类型、Python 中的类以及 JavaScript 中的对象。结构体把各种类型的值聚集为一个值，以便作为一个整体来处理。给定一个结构体，我们可以读取或修改它的任意组件。结构体也可以有关联的方法，用于操作自己的组件。

Rust 有 3 种结构体类型：**命名字段**（named-field）**结构体**、**类元组**（tuple-like）**结构体**和**类基元**（unit-like）**结构体**，区别在于如何引用它们的组件。命名字段结构体的每个组件都有一个名字，类元组结构体以组件出现的次序来标识它们，类基元结构体则根本没有组件。类基元结构体并不常见，但比我们想象的有用。

本章将详细介绍每一种结构体，展示它们在内存中的样子。还将讨论如何给它们添加方法、如何定义对不同组件类型都有效的泛型结构体，以及如何让 Rust 为结构体生成常用特型（trait）的实现。

9.1　命名字段结构体

下面是一个命名字段结构体类型的定义：

```
/// 8位灰阶像素的矩形
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

这里声明的结构体类型为 GrayscaleMap，它包含两个给定类型的字段：pixels 和 size。Rust 对所有类型（包括结构体）的命名有一个约定，即每个单词的首字母要大写，比如 GrayscaleMap，称为**驼峰拼写法**（CamelCase）。字段和方法的名字要小写，单词间以下划线分隔，称为**蛇形拼写法**（snake_case）。

可以使用**结构体表达式**（struct expression）创建结构体的值，比如：

```
let width = 1024;
let height = 576;
let image = GrayscaleMap {
    pixels: vec![0; width * height],
    size: (width, height)
};
```

结构体表达式以类型名（GrayscaleMap）开头，后跟一对花括号，其中列出了每个字段的名字和值。如果局部变量或参数与字段同名，那么也可以采用以下函数中的简写形式：

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) -> GrayscaleMap {
    assert_eq!(pixels.len(), size.0 * size.1);
    GrayscaleMap { pixels, size }
}
```

函数中的结构体表达式 GrayscaleMap { pixels, size } 是对 GrayscaleMap { pixels: pixels, size: size } 的简写。在同一个结构体表达式中，可以同时对某些字段使用 key: value 语法，而对另一些字段使用简写语法。

使用熟悉的 . 操作符访问结构体的字段：

```
assert_eq!(image.size, (1024, 576));
assert_eq!(image.pixels.len(), 1024 * 576);
```

与 Rust 中的其他构成项一样，结构体默认是私有的，其只在声明它的模块中可见。要想让结构体对模块外部可见，需要在它的定义之前加上 pub 关键字。结构体中的字段默认也是私有的：

```
/// 8位灰阶像素的矩形
pub struct GrayscaleMap {
    pub pixels: Vec<u8>,
    pub size: (usize, usize)
}
```

就算结构体声明为公有（pub），其字段仍然可以私有：

```
/// 8位灰阶像素的矩形
pub struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

其他模块可以使用这个结构体以及它的任何公有方法，但不能通过名字访问其私有字段，也不能使用结构体表达式创建新 GrayscaleMap 值，也就是说，创建结构体值要求结构体的所有字段都必须是可见的。这也是不能通过结构体表达式来创建新 String 或 Vec 的原因。这些标准类型都是结构体，但它们的字段都是私有的。如果想创建它们的值，则必须使用

其公有方法，比如 Vec::new()。

创建命名字段结构体的值时，可以使用另一个相同类型的结构体来提供省略的字段值。在结构体表达式中，如果命名字段后面跟着 .. EXPR，那么任何没有出现的字段都将从 EXPR 中取得自己的值。当然，前提是 EXPR 必须为同一结构体类型的另一个值。假设有下面这个表示游戏中怪物的结构体：

```
struct Broom {
    name: String,
    height: u32,
    health: u32,
    position: (f32, f32, f32),
    intent: BroomIntent
}

/// 扫帚（Broom）可能干的两件事儿
#[derive(Copy, Clone)]
enum BroomIntent { FetchWater, DumpWater }
```

2010 年上映的美国奇幻电影 *The Sorcerer's Apprentice*（《魔法师的学徒》）中，学徒给一把扫帚施了魔法，让它帮自己干活儿，可是活儿干完了却不知道怎么让扫帚停下来。他用斧子把扫帚一砍两截，结果一把扫帚变成了两把短扫帚，照旧尽职尽责地干活儿。

```
// 按值接收Broom，取得所有权
fn chop(b: Broom) -> (Broom, Broom) {
    // 基于b初始化broom1，只修改height。因为String不是可复制类型，
    // 所以broom1取得b名字的所有权
    let mut broom1 = Broom { height: b.height / 2, .. b };

    // 基于broom1初始化broom2。因为String不是可复制类型，
    // 所以必须显式地克隆name
    let mut broom2 = Broom { name: broom1.name.clone(), .. broom1 };

    // 给每一截“怪物”扫帚起个不一样的名字
    broom1.name.push_str(" I");
    broom2.name.push_str(" II");

    (broom1, broom2)
}
```

有了以上定义，接下来就可以创建一把扫帚（broom），把它一砍两截，看看结果如何：

```
let hokey = Broom {
    name: "Hokey".to_string(),
    height: 60,
    health: 100,
    position: (100.0, 200.0, 0.0),
    intent: BroomIntent::FetchWater
};

let (hokey1, hokey2) = chop(hokey);
assert_eq!(hokey1.name, "Hokey I");
assert_eq!(hokey1.health, 100);

assert_eq!(hokey2.name, "Hokey II");
assert_eq!(hokey2.health, 100);
```

9.2　类元组结构体

第二种结构体类型叫**类元组结构体**，因为它类似元组：

```
struct Bounds(usize, usize);
```

创建这种类型的值就像创建元组一样，只不过要加上结构体的名字：

```
let image_bounds = Bounds(1024, 768);
```

类元组结构体的值称为**元素**（element），跟元组的值一样。访问这些值同样也跟访问元组的值一样：

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

类元组结构体中个别的元素可以是公有的，也可以不是：

```
pub struct Bounds(pub usize, pub usize);
```

表达式 `Bounds(1024, 768)` 看起来像个函数调用，而实际上也是。定义这种结构体会隐式定义一个函数：

```
fn Bounds(elem0: usize, elem1: usize) -> Bounds { ... }
```

本质上，命名字段结构体与类元组结构体非常相似。选择用哪一种需要考虑易读性、歧义性和简洁性。如果很多时候要使用 `.` 操作符取得值的组件，则名字标识的字段能为读者提供更多信息，也更不容易写错。如果经常需要使用模式匹配来查询元素，那类元组结构体更合适。

类元组结构体很适合创建**新类型**（newtype），即只包含一个要经过更严格类型检查的组件的结构体。比如，如果想只使用 ASCII 文本，那么可以像这样定义一个新类型：

```
struct Ascii(Vec<u8>);
```

与传递 `Vec<u8>` 内存缓冲区并通过注释说明它们是什么相比，用上面这个类型来声明 ASCII 字符串要明确得多。这个新类型可以帮助 Rust 发现错误，比如给一个接收 ASCII 文本的函数传递了其他字节缓冲区。第 21 章将介绍一个使用新类型实现高效类型转换的例子。

9.3　类基元结构体

第三种结构体有点不好理解，因为这是一种完全没有元素的结构体：

```
struct Onesuch;
```

这种类型的值不占内存，非常像基元类型 `()`。Rust 不会把类基元结构体的值保存到内存里，也不会生成操作它们的代码，因为通过类型就知道它的值了。从逻辑上讲，空结构体类型的值是一样的，或者更准确地说，这种类型都只有一个值：

```
let o = Onesuch;
```

在前面 6.9 节中，我们曾碰到过一个类基元结构体。表达式 3..5 是对结构体值 Range { start: 3, end: 5 } 的简写形式，而表达式 ..（省略了两个端点的范围）是对类基元结构体值 RangeFull 的简写。

类基元结构体在使用特型的时候也有用，第 11 章将讨论这一点。

9.4 结构体布局

在内存中，命名字段结构体与类元组结构体一样都是值的集合，类型可以不同，并以特定方式存储。比如，本章前面定义的结构体 GrayscaleMap：

```
struct GrayscaleMap {
    pixels: Vec<u8>,
    size: (usize, usize)
}
```

这个 GrayscaleMap 的值在内存中的布局如图 9-1 所示。

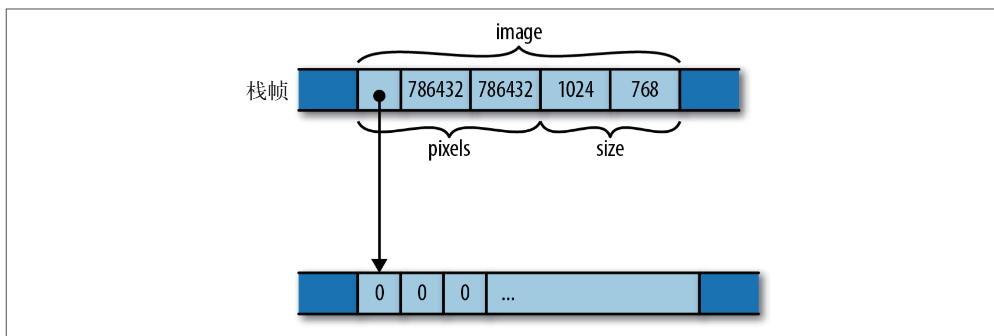


图 9-1：GrayscaleMap 结构体在内存中的存储方式

与 C 和 C++ 不同，Rust 不保证结构体的字段或元素在内存中会以某种顺序存储，上面的示意图只展示了一种可能的布局。不过，Rust 保证把字段的值直接存储在结构体的内存块中。JavaScript、Python 和 Java 会把 pixels 和 size 的值分别存储到各自在堆内存上分配到的块中并使 GrayscaleMap 字段指向它们，而 Rust 直接把 pixels 和 size 放到 GrayscaleMap 值的内存里，只有 pixels 向量拥有自己分配在堆上的内存块。

可以使用 #[repr(C)] 属性要求 Rust 以兼容 C 和 C++ 的方式在内存中存储结构体。第 21 章将详细讨论这一点。

9.5 通过impl定义方法

到目前为止，本书已经展示了在各种值上调用多种方法的例子。比如，调用 v.push(e) 把元素推到向量里、调用 v.len() 取得向量的长度、调用 r.expect("msg") 检查 Result 值是不是包含错误，等等。

可以给任意结构体类型定义方法。但与 C++ 或 Java 直接在结构体中定义方法不同，在 Rust 中定义方法要使用单独的 impl 块，比如：

```
/// 一个后进先出的字符队列
pub struct Queue {
    older: Vec<char>,   // 旧元素，最老的在最后
    younger: Vec<char>  // 新元素，最新的在最后
}

impl Queue {
    /// 把一个字符推到队列后端
    pub fn push(&mut self, c: char) {
        self.younger.push(c);
    }

    /// 从队列前端取出一个字符，如果可以，取出字符返回Some(c),
    /// 否则如果队列是空的，返回None
    pub fn pop(&mut self) -> Option<char> {
        if self.older.is_empty() {
            if self.younger.is_empty() {
                return None;
            }

            // 把younger中的元素转移到older中,
            // 并保持对外承诺的顺序
            use std::mem::swap;
            swap(&mut self.older, &mut self.younger);
            self.older.reverse();
        }

        // 到这里older肯定有元素。Vec的pop方法已经返回Option，这就可以了
        self.older.pop()
    }
}
```

impl 块只是 fn 定义的集合，这些函数都会成为块顶部提到名字的那个结构体类型的方法。这里先定义了一个公有结构体 Queue，接着又给它定义了两个公有方法 push 和 pop。

方法也称为**关联函数**（associated funtion），因为它们是与特定类型关联的。与关联函数相对的叫**自由函数**（free function），即不是作为 impl 块中的构成项定义的函数。

Rust 作为第一个参数传入并且要在其上调用这个方法的值必须有一个特殊的名字：self。因为 self 的类型明显就是 impl 块顶部提到名字的那个结构体或者对那个结构体的引用，所以 Rust 允许省略它的类型声明。而使用简写的 self、&self 和 &mut self 分别代表 self: Queue、self: &Queue 和 self: &mut Queue。如果你愿意，也可以使用包含类型声明的完整形式，但几乎所有 Rust 代码都会像上面一样使用简写。

在前面的例子中，push 和 pop 方法通过 self.older 和 self.younger 引用 Queue 的字段。我们知道，在 C++ 和 Java 的方法体中，可以通过非限定标识符直接访问“这个”（this）对象的成员，而 Rust 方法必须显示使用 self 来引用调用该方法时作为上下文的那个值，这与 Python 方法使用 self 以及 JavaScript 方法使用 this 的方式类似。

由于 push 和 pop 要修改 Queue，因此它们的参数都是 &mut self。不过，调用方法时不需要借用这个可修改的引用，因为普通的方法调用语法会自动实现隐式借用。有了以上方法定义，就可以像下面这样来使用 Queue 了：

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('0');
q.push('1');
assert_eq!(q.pop(), Some('0'));

q.push('∞');
assert_eq!(q.pop(), Some('1'));
assert_eq!(q.pop(), Some('∞'));
assert_eq!(q.pop(), None);
```

只要写 `q.push(...)` 就能借用一个对 `q` 的可修改引用，就像你写的是 `(&mut q).push(...)` 一样，因为 `push` 的 `self` 参数就是这么要求的。

如果方法不需要修改其 `self`，那么可以让它接收一个共享引用。比如：

```
impl Queue {
    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }
}
```

同样，方法调用表达式知道要借用哪种引用：

```
assert!(q.is_empty());
q.push('☉');
assert!(!q.is_empty());
```

如果方法要取得 `self` 的所有权，则可以取得 `self` 的值：

```
impl Queue {
    pub fn split(self) -> (Vec<char>, Vec<char>) {
        (self.older, self.younger)
    }
}
```

调用这个 `split` 方法看起来与调用其他方法相似：

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('P');
q.push('D');
assert_eq!(q.pop(), Some('P'));
q.push('X');

let (older, younger) = q.split();
// q现在是未初始化状态
assert_eq!(older, vec!['D']);
assert_eq!(younger, vec!['X']);
```

但要注意，由于 `split` 取得了 `self` 的值，从而把 `Queue` **转移**出了 `q`，导致 `q` 变成了未初始化状态。正因为 `split` 中的 `self` 现在拥有了这个队列，所以它才能把个别的向量转移出来并返回给调用者。

还可以定义根本不将 `self` 作为参数的方法。这样的方法就成了与结构体类型本身而非该类

型的值关联的函数。遵循 C++ 和 Java 的传统，Rust 称这些方法为**静态方法**。静态方法通常用于定义构造器函数，比如：

```
impl Queue {
    pub fn new() -> Queue {
        Queue { older: Vec::new(), younger: Vec::new() }
    }
}
```

引用这个方法的方式是 `Queue::new`，即类型名、双冒号和方法名。如此一来，示例代码就可以变得更简洁了：

```
let mut q = Queue::new();

q.push('*');
...
```

把构造器函数命名为 `new` 是 Rust 的惯例，前面已经见过的有 `Vec::new`、`Box::new`、`HashMap::new`，等等。但 `new` 这个名字本身没什么特别的，它不是关键字。很多类型会使用其他名字命名的静态方法作为构造器，比如 `Vec::with_capacity`。

虽然一个类型可以对应多个 `impl` 块，但这些块必须全部位于定义该类型的同一个 Rust 包中。Rust 允许开发者给其他类型附加自己的方法，第 11 章将具体介绍。

如果你熟悉 C++ 或 Java，可能会觉得将类型与方法的定义分开有点异乎寻常，但这样做有以下几个好处。

- 容易找到类型的数据成员。在一个很大的 C++ 类定义中，可能需要浏览几百行代码的成员函数定义，才能对这个类的所有数据成员有个大致的了解。而在 Rust 中，它们都在一个地方。
- 尽管可以考虑将方法定义与命名字段结构体的语法结合起来，但对于类元组结构体和类基元结构体而言就会显得不够简洁。把方法抽取到 `impl` 块中，可以实现一种语法兼顾这 3 种结构体。事实上，Rust 也使用相同的语法给非结构体类型，比如 `enum` 类型和 `i32` 这种基本类型，定义方法。（正是由于所有类型都可以有方法，因此 Rust 才很少使用**对象**这个术语，而是更倾向于把一切都称为**值**。）
- 同样的 `impl` 语法也能轻松地用于实现特型，第 11 章将探讨这一块。

9.6 泛型结构体

前面对 `Queue` 的定义不够令人满意。那样的写法只能存储字符，但它的结构或方法并非只能用于字符。如果我们还定义了另一个结构体，用于存储比如 `String` 值，那么除了用 `String` 代替 `char` 之外，其他代码可以都不改。但这不是在浪费时间吗？

好在 Rust 结构体可以是**泛型**（generic）的，也就是说，这种结构体是一个模板，可以在其中插入任何类型。比如，下面定义的这个 `Queue` 就可以存储任何类型的值：

```
pub struct Queue<T> {
```

```
    older: Vec<T>,
    younger: Vec<T>
}
```

可以把 Queue<T> 中的 <T> 读作“对于任何元素类型 T……”。因此，上面的定义就可以翻译成：“对于任意类型 T，Queue<T> 包含两个 Vec<T> 类型的字段。”比如，对于 Queue<String>，T 是 String，因此 older 和 younger 的类型都是 Vec<String>。而对于 Queue<char>，T 是 char，那么这个结构体就跟开始定义的专门针对 char 的结构体一样。事实上，Vec 本身也是一个泛型结构体，它就是以这种方式定义的。

在泛型结构体的定义中，位于尖括号（<>）中的“类型名”叫作**类型参数**。与泛型结构体对应的 impl 块类似如下所示：

```
impl<T> Queue<T> {
    pub fn new() -> Queue<T> {
        Queue { older: Vec::new(), younger: Vec::new() }
    }

    pub fn push(&mut self, t: T) {
        self.younger.push(t);
    }

    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }

    ...
}
```

第一行中的 impl<T> Queue<T> 可以理解为：“对于任意类型 T，这里给 Queue<T> 定义几个方法”。之后就可以在方法定义中使用类型参数 T 作为一种类型了。

前面的代码中使用过 Rust 简写的 self 参数。这么一比，到处写 Queue<T> 则显得多余又容易让人分心。为此，impl 块（无论是不是泛型）还定义了一个特殊类型参数 Self（注意驼峰式拼写的名字），表示要把方法添加到其中的任意类型。这也是一种简写形式。对前面的代码而言，Self 就表示 Queue<T>，因此可以让 Queue::new 的定义再简短一点：

```
pub fn new() -> Self {
    Queue { older: Vec::new(), younger: Vec::new() }
}
```

可能读者也注意到了，在 new 的方法体中，不需要在构造表达式内写出类型参数，只写 Queue { ... } 就可以。Rust 的类型推断会起作用：既然只有一种类型即 Queue<T> 适合作为该函数的返回值，那 Rust 会替我们给出这个参数。但是，在函数签名和类型定义中仍然需要给出类型参数。Rust 不会推断这些情况下的参数类型，相反，它要根据这些地方显式给出的类型来推断函数体内的类型。

调用静态方法时，可以使用“极速鱼”符号 ::<> 显示地提供类型参数：

```
let mut q = Queue::<char>::new();
```

但在实际开发中，通常让 Rust 为你推断就好了：

```
let mut q = Queue::new();
let mut r = Queue::new();

q.push("CAD");  // 明显是Queue<&'static str>
r.push(0.74);   // 明显是Queue<f64>

q.push("BTC");  // 比特币美元价格，2017年5月
r.push(2737.7); // Rust检测不出非理性繁荣
```

事实上，本书在使用 `Vec` 这个泛型结构体时始终都是这么做的。

不仅结构体可以是泛型的，枚举类型也可以接收类型参数，而且语法还非常相似。第 12 章将详细介绍枚举（`enum`）类型。

9.7 带生命期参数的结构体

正如 5.2.5 节讨论的，如果结构体类型包含引用，则必须指定这些引用的生命期。比如，下面这个结构体可以包含对某个片段（slice）中最大和最小元素的引用：

```
struct Extrema<'elt> {
    greatest: &'elt i32,
    least: &'elt i32
}
```

前面说过，可以把 `struct Queue<T>` 这样的声明理解为：给定任意类型 `T`，都可以创建一个包含该类型的 `Queue<T>`。类似地，可以把 `struct Extrema<'elt>` 理解为：给定任意生命期 `'elt`，都可以创建一个包含具有该生命期引用的 `Extrema<'elt>`。

下面这个函数会检查一个片段并返回一个 `Extrema` 值，该值的字段会引用片段的元素：

```
fn find_extrema<'s>(slice: &'s [i32]) -> Extrema<'s> {
    let mut greatest = &slice[0];
    let mut least = &slice[0];

    for i in 1..slice.len() {
        if slice[i] < *least    { least    = &slice[i]; }
        if slice[i] > *greatest { greatest = &slice[i]; }
    }
    Extrema { greatest, least }
}
```

这里，由于 `find_extrema` 借用了 `slice` 的元素，而 `slice` 的生命期为 `'s`，因此返回的结构体 `Extrema` 也会使用 `'s` 作为其引用的生命期。Rust 会推断函数调用的生命期参数，所以调用 `find_extrema` 时不需要指出它们：

```
let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);
```

鉴于返回的类型经常使用与某个参数相同的生命期，Rust 允许在明显有候选项时省略生命

期。为此，find_extrema 的签名其实可以写成下面这样，含义不变：

```
fn find_extrema(slice: &[i32]) -> Extrema {
    ...
}
```

没错，我们的意思也**可能**是 Extrema<'static>，但那种情况太少见了。Rust 为常见情况提供简写形式。

9.8 为结构体类型派生共有特型

结构体可以很简单：

```
struct Point {
    x: f64,
    y: f64
}
```

然而，如果你打算使用这个 Point 类型，又会发现有点麻烦。根据定义，Point 不可以复制、不可以克隆，也不能通过 println!("{:?}", point) 打印，还不支持 == 和 != 操作符。

上述每个特性在 Rus 中都有一个名字：Copy、Clone、Debug 和 PartialEq，它们被称为**特型**（trait）。第 11 章将探讨如何手工为自定义结构体实现特型。但对于这些标准的特型（以及其他一些特型），是不需要手工实现的，除非你需要额外自定义行为。Rust 可以自动帮你实现它们，分毫不差。为此，只需给结构体添加一个 #[derive] 属性：

```
#[derive(Copy, Clone, Debug, PartialEq)]
struct Point {
    x: f64,
    y: f64
}
```

只要结构体的每个字段都实现了这些特型，结构体就能自动实现它们。比如，之所以可以让 Rust 为 Point 派生 PartialEq，是因为它的两个字段都是 f64 类型的，而该类型已经实现了 PartialEq。

Rust 也可以派生 PartialOrd，从而额外支持 <、>、<=、>= 等比较操作符。这里并没有这么做，因为通过比较两个点来看其中一个是不是“小于”另一个意义不大。毕竟点和点之间没有事先约定的次序。所以此处选择不让 Point 值支持这些操作符。正是由于存在类似这种情况，因此 Rust 并没有自动派生所有特型，而是让我们自己通过 #[derive] 属性来指定。另外一个原因是实现的特型会自动成为结构体的公有特性，也就是说复制、克隆等会成为结构体的公有 API，而这是需要慎重考虑的。

第 13 章将详细介绍 Rust 的标准特型，以及哪些特型可以通过 #[derive] 属性来派生。

9.9 内部修改能力

修改能力（mutability）与其他东西一样：过度使用会出问题，而你经常只需要一小点就够了。比如，你的爬虫机器人控制系统有一个中心化的结构体，叫 SpiderRobot，包含设置

及 I/O 勾柄。机器人启动时会创建这个结构体，值永远不变：

```
pub struct SpiderRobot {
    species: String,
    web_enabled: bool,
    leg_devices: [fd::FileDesc; 8],
    ...
}
```

这个机器人的每个主系统都由一个不同的结构体把控，且它们都包含一个指向 `SpiderRobot` 的指针：

```
use std::rc::Rc;

pub struct SpiderSenses {
    robot: Rc<SpiderRobot>,   // <-- 指向设置与I/O的指针
    eyes: [Camera; 32],
    motion: Accelerometer,
    ...
}
```

其他诸如 Web 构建、捕食、毒液流控制之类的结构体中，也都包含一个 `Rc<SpiderRobot>` 智能指针。还记得吧，`Rc` 代表“reference counting”（引用计数），而 `Rc` 盒子中的值始终是共享的，因此永远不能修改。

现在假设你想使用标准的 `File` 类型在 `SpiderRobot` 结构体中记录一些信息。有一个问题：`File` 必须是 `mut`，所有向它写入信息的方法都需要一个对它的 `mut` 引用。

类似的情况很常见。这时候你需要的只是在一个不可修改的值（`SpiderRobot` 结构体）内部有一个小小的可修改数据（一个 `File`）。这就叫作**内部修改能力**（interior mutability）。Rust 为此提供了多种支持方式，本节只讨论两种最简单的类型，即 `Cell<T>` 和 `RefCell<T>`，它们都在 `std::cell` 模块中定义。

`Cell<T>` 是只包含一个 `T` 类型私有值的结构体。`Cell` 唯一特别的地方是不需要对其自身的 `mut` 引用，你也能取得或设置其私有字段的值。

- **`Cell::new(value)`**：创建一个新 `Cell`，将 `value` 转移到其中。
- **`cell.get()`**：返回 `cell` 中值的副本。
- **`cell.set(value)`**：把 `value` 保存到 `cell`，丢弃之前保存的值。

这个方法的 `self` 参数是以非 `mut` 引用的形式传入的：

```
fn set(&self, value: T)     // 注意：不是&mut self
```

当然，对于命名为 `set` 的方法来说，这是不常见的。到目前为止，Rust 给我们的感觉是只要修改数据，就得获取 `mut` 权限。但同样地，这个不同寻常的细节也正是 `Cell` 的关键所在。`Cell` 仅仅是恰到好处地为违背不可修改规则提供了一种安全的方式。

`Cell` 还有其他一些方法，具体可以查阅相关文档。

如果只是给 `SpiderRobot` 添加一个简单的计数器，那么用 `Cell` 很方便。可以这么写：

```
use std::cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count: Cell<u32>,
    ...
}
```

然后，即使是 `SpiderRobot` 的非 `mut` 方法也可以通过 `.get()` 和 `.set()` 方法读写这个 `u32` 值：

```
impl SpiderRobot {
    /// 错误数加1
    pub fn add_hardware_error(&self) {
        let n = self.hardware_error_count.get();
        self.hardware_error_count.set(n + 1);
    }

    /// 只要发生过硬件错误就是true
    pub fn has_hardware_errors(&self) -> bool {
        self.hardware_error_count.get() > 0
    }
}
```

非常简单，可这不能解决我们记录信息的问题。`Cell` **不让**在共享的值上调用 `mut` 方法。`.get()` 方法返回 `Cell` 中值的副本，因此 `T` 必须实现 `Copy` 特型。为了记录信息，需要一个可修改的 `File`，而 `File` 不可复制。

此时正确的选择是 `RefCell`。与 `Cell<T>` 类似，`RefCell<T>` 是只包含一个 `T` 类型值的泛型类型。但与 `Cell` 不同，`RefCell` 支持借用它的 `T` 类型值的引用。

- **`RefCell::new(value)`**：创建一个新 `RefCell`，将 `value` 转移到其中。
- **`ref_cell.borrow()`**：返回一个 `Ref<T>`，基本上是对 `ref_cell` 中值的共享引用。
  如果这个值已经被可修改地借用了，这个方法会诧异，详细信息见下文。
- **`ref_cell.borrow_mut()`**：返回一个 `RefMut<T>`，基本上是对 `ref_cell` 中值的可修改引用。
  如果这个值已经被借用了，这个方法会诧异，详细信息见下文。

同样，`RefCell` 也有其他方法，可以查阅相关文档。

上述两个 `borrow` 方法之所以会诧异，是因为你试图破坏 Rust 中 `mut` 引用是排他引用的规则。比如，以下代码会引发诧异：

```
let ref_cell: RefCell<String> = RefCell::new("hello".to_string());

let r = ref_cell.borrow();          // 没问题，返回一个Ref<String>
let count = r.len();                // 没问题，返回"hello".len()
assert_eq!(count, 5);

let mut w = ref_cell.borrow_mut();  // 诧异：已经被借用了
w.push_str(" world");
```

为避免诧异，可以把这两次借用放到两个独立的块中。这样，`r` 会在你借用 `w` 之前被丢弃。

这跟常规引用的工作方式非常相似。唯一的不同是，常规情况下把引用保存到变量，Rust

会通过**编译时**检查来确保安全地使用它。如果检查失败，编译器会报错。而 `RefCell` 会通过运行时检查强制应用相同的规则。因此如果你破坏了规则，就会收到诧异。

现在可以在 `SpiderRobot` 类型中使用 `RefCell` 了：

```
pub struct SpiderRobot {
    ...
    log_file: RefCell<File>,
    ...
}

impl SpiderRobot {
    /// 向日志文件中写入一行信息
    pub fn log(&self, message: &str) {
        let mut file = self.log_file.borrow_mut();
        writeln!(file, "{}", message).unwrap();
    }
}
```

变量 `file` 的类型是 `RefMut<File>`，可以像使用 `File` 的可修改引用一样使用它。第 18 章将详细讲解写入文件的相关内容。

`Cell` 和 `RefCell` 很方便，就是调用 `.get()`、`.set()` 或 `.borrow()`、`.borrow_mut()` 稍微麻烦点，但这也只是对违背规则的一个小小的惩罚。有一个缺点没那么明显，却很严重：`Cell` 和 `RefCell`，以及包含它们的任何类型，都不是线程安全的。为此，Rust 不会同时允许多个线程访问它们。第 19 章在讨论到"`Mutex<T>`"（参见 19.3.2 节）、"原子类型"（参见 19.3.10 节）和"全局变量"（参见 19.3.11 节）时将介绍线程安全的内部修改能力。

一个结构体，无论它包含命名字段还是类元组形式，都是其他值的集合体。如果我有一个 `SpiderSenses` 结构体，那就有了指向共享 `SpiderRobot` 结构体的 `Rc` 指针（`robot: Rc<SpiderRobot>`）、有了"眼睛"（`eyes: [Camera; 32]`）、有了加速计（`motion: Accelerometer`），等等。因此结构体的本质就是一个"和"字：我有 *X* 和 *Y*。有没有一种构建在"或"字基础上的类型呢？换句话说，如果你有一个那种类型的值，那么你**或者**有 *X*，**或者**有 *Y*。这样的类型也非常有用，而它们在 Rust 中几乎无所不在，下一章将讨论这种类型。

第 10 章

枚举与模式

非常奇怪要多少计算机知识才能解释为什么“总和”类型（sum type）如此稀缺。（参照：稀缺的兰布达）

——Graydon Hoare

本章的第一个主题非常有势力，且可谓源远流长，它能迅速帮你“摆平很多事”（有点代价）。不同文化背景下对它的称呼各不相同，但它不干坏事。它是一种用户定义的数据类型。很早以前，ML 和 Haskell 程序员就叫它“总和”（sum）类型、“有别联合”（discriminated union）或“代数数据”（algebraic data）类型。Rust 称其为**枚举**，即 `enum`。枚举类型不仅不干坏事，还很安全，它所要求的代价是不能有大的损失。

C++ 和 C# 都有枚举，通过它可以定义一个你自己的类型，值是一组命名常量。比如，定义一个名为 `Color` 的类型，值为 `Red`、`Orange`、`Yellow` 等。Rust 也可以定义这样的枚举，且不止如此，Rust 枚举还可以包含数据，而且是不同类型的数据。比如，Rust 的 `Result<String, io::Error>` 是一个枚举类型，它的值要么是一个包含 `String` 的 `Ok` 值，要么是一个包含 `io::Error` 的 `Err` 值。C++ 和 C# 无法定义这样的枚举。它更像是 C 的 `union`，但不同的是 Rust 枚举是类型安全的。

如果一个值有几种可能的结果，那使用枚举来表示会比较方便。使用枚举的“代价”是必须保证安全地访问数据，也就是必须使用模式匹配，这是本章后一半的主题。

知道 Python 中的解包（unpack）或 JavaScript 中的解构（destructure），自然就知道模式了。但 Rust 中的模式同样不止如此，它有点像能够匹配任意数据的正则表达式。可以用模式测试某个值是否具有特定格式，或者一次性把结构体或元组的多个字段提取到本地变量中。模式还有一点跟正则表达式很像：非常简洁，通常只要一行代码即可达到目的。

10.1　枚举

定义 C 式枚举很简单：

```
enum Ordering {
    Less,
    Equal,
    Greater
}
```

这里声明的 `Ordering` 类型包含 3 个可能的值，名为**变体**或**构造式**（constructor）：`Ordering::Less`、`Ordering::Equal` 和 `Ordering::Greater`。这个枚举是标准库里定义的，因此可以在 Rust 代码中导入它，或者只导入枚举本身：

```
use std::cmp::Ordering;

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
       Ordering::Less
    } else if n > m {
       Ordering::Greater
    } else {
       Ordering::Equal
    }
}
```

或者同时导入它的所有构造式：

```
use std::cmp::Ordering;
use std::cmp::Ordering::*;    // *用于导入所有子元素

fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Less
    } else if n > m {
        Greater
    } else {
        Equal
    }
}
```

导入构造式后，就可以不写 `Ordering::Less` 而只写 `Less` 了，但这样意思就没那么明确了。一般认为，除非可以让代码更好理解，否则**不要**导入构造式。

要导入当前模块中声明的枚举的构造式，使用 `self`：

```
enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;
```

在内存中，C 式枚举的值存储为整数。有时候可能要告诉 Rust 使用什么整数：

```
enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}
```

否则，Rust 会替你从 0 开始赋值。

Rust 默认使用能够容纳它们的最小内置整数类型来存储 C 式枚举。多数情况下一个字节就够了。

```
use std::mem::size_of;
assert_eq!(size_of::<Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2);    // 一个u8装不下404
```

可以通过在定义枚举时添加 `#[repr]` 属性来改变 Rust 在内存中存储枚举的方式，相关细节参见第 21 章。

Rust 允许把 C 式枚举值转换为整数：

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

但不能直接把整数转换为枚举值。与 C 和 C++ 不同，Rust 要保证枚举值一定是在 `enum` 声明中定义的。不经验证的整数类型到枚举类型的转换无法保证这一点。除非你自己写一个带验证的转换：

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None
    }
}
```

或者使用 `enum_primitive` 包，其中有一个宏能帮你自动生成这种转换代码。

跟结构体一样，编译器也能给枚举添加类似 `==` 操作符这样的特性，但必须使用 `#[derive]` 属性来声明。

```
#[derive(Copy, Clone, Debug, PartialEq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years
}
```

枚举也可以有方法，同样跟结构体一样：

```
impl TimeUnit {
    /// 返回这个时间单位的复数名词
    fn plural(self) -> &'static str {
        match self {
```

```
            TimeUnit::Seconds => "seconds",
            TimeUnit::Minutes => "minutes",
            TimeUnit::Hours => "hours",
            TimeUnit::Days => "days",
            TimeUnit::Months => "months",
            TimeUnit::Years => "years"
        }
    }

    /// 返回这个时间单位的单数名词
    fn singular(self) -> &'static str {
        self.plural().trim_right_matches('s')
    }
}
```

C 式枚举介绍的差不多了。Rust 枚举更有意思的地方在于它能包含的数据类型。

10.1.1　包含数据的枚举

有的程序需要显示精确到毫秒的完整日期和时间，但对多数应用而言，显示一个大致的近似值（比如“两个月前”）对用户更友好。可以写一个枚举来实现这种转换：

```
/// 生成向上进位的时间戳，让程序显示“6个月前”，
/// 而不是“2020年4月9日晚上10点55分”
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32)
}
```

这个枚举中的两个变体 `InThePast` 和 `InTheFuture` 都接收参数。这种变体叫**元组变体**（tuple variant）。与元组结构体一样，这些构造式都是用于生成新 `RoughTime` 值的函数。

```
let four_score_and_seven_years_ago =
    RoughTime::InThePast(TimeUnit::Years, 4*20 + 7);

let three_hours_from_now =
    RoughTime::InTheFuture(TimeUnit::Hours, 3);
```

枚举值也可以是**结构体变体**（struct variant），与常规结构体一样包含命名字段：

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
    Cuboid { corner1: Point3d, corner2: Point3d }
}

let unit_sphere = Shape::Sphere { center: ORIGIN, radius: 1.0 };
```

总之，Rust 有 3 种枚举变体，分别对应上一章介绍的 3 种结构体。没有数据的变体对应类基元结构体。元组变体的写法和作用与元组结构体一样。结构体变体则有花括号和命名字段。一个枚举可以同时包含所有这 3 种变体。

```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
        car: DifferentialEquation,
        cdr: EarlyModernistPoem
    }
}
```

公有枚举的所有构造式和字段自动是公有的。

10.1.2　枚举的内存布局

在内存中，带数据的枚举的每个构造式都需要一个小整数**标签**（tag）和足以容纳最大变体所有字段的内存来存储。整数标签是 Rust 内部使用的字段，通过它可以知道是哪个构造式创建了当前的值，以及当前的值包含哪些字段。

截止到 Rust 1.17，`RoughTime` 的每个构造式占用 8 个字节，如图 10-1 所示。

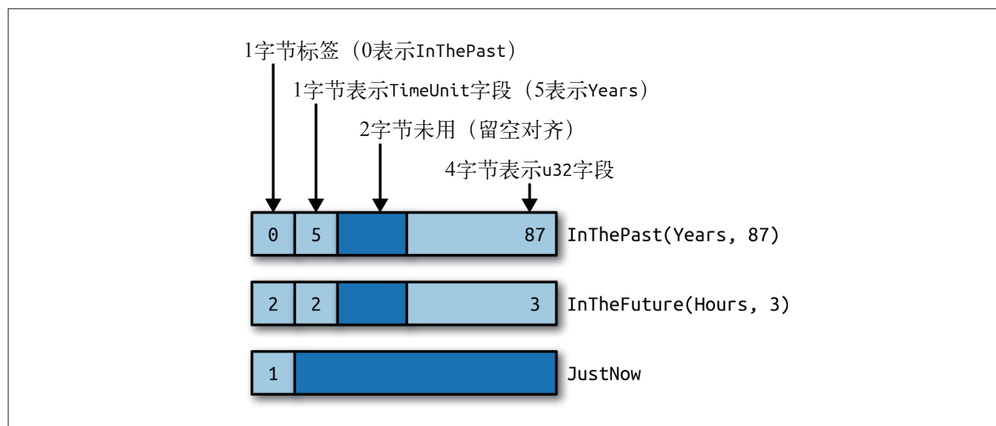


图 10-1：`RoughTime` 值在内存中的布局

为方便未来的优化，Rust 并未对枚举的内存布局方式做出任何承诺。有些情况下可能存在比图 10-1 更高效的存储方式。正如本章后面会提到的，Rust 对某些枚举会直接优化掉标签字段。

10.1.3　使用枚举的富数据结构

枚举也可以用来快速实现类似树的数据结构。比如，假设有一个 Rust 程序需要使用任意 JSON 数据。在内存中，任何 JSON 文档都可以用这个 Rust 类型的值表示：

```
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
```

```
        String(String),
        Array(Vec<Json>),
        Object(Box<HashMap<String, Json>>)
    }
```

以上 Rust 代码本身就足以说明这个数据结构了，即使人类语言的表达能力也很难超越。JSON 标准规定了可以出现在 JSON 文档中的不同数据类型：`null`、布尔值、数值、字符串、JSON 值数组，以及以字符串为键以 JSON 值为值的对象。上面这个 `Json` 枚举已经把这些类型表达得很清楚了。

这不是一个假想的例子。`serde_json` 是 crates.io 上下载量最大的库之一，作为一个实现 Rust 结构体序列化的库，它的实现中就包含一个与上例极为相似的枚举定义。

用 `Box` 包装 `HashMap` 来表示 `Object` 只是为了让所有 `Json` 值更简洁。在内存中，`Json` 类型的值占 4 个机器字。`String` 和 `Vec` 值占 3 个机器字，Rust 还会加一个标签字节。`Null` 和 `Boolean` 值用不了那么多内存空间，但所有 `Json` 值的大小必须相同。因此剩余空间就用不上了。图 10-2 给出了 `Json` 值在内存中布局的一个例子。

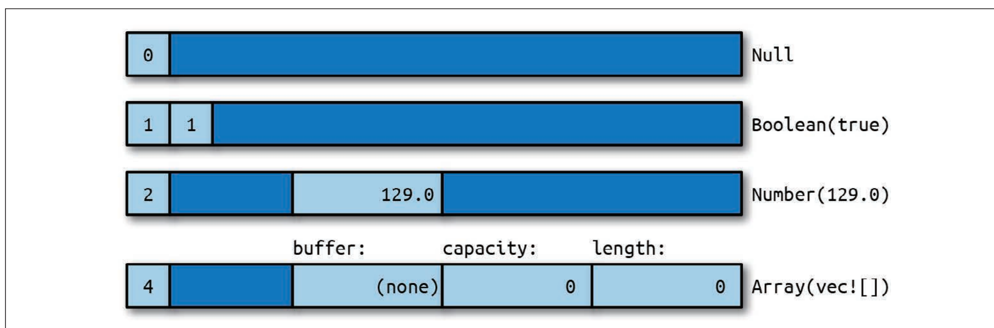


图 10-2：`Json` 值在内存中的布局

`HashMap` 其实还要更大。如果每个 `Json` 值都考虑要容纳它，那就太大了，大概得 8 个机器字。但 `Box<HashMap>` 只占 1 个机器字，因为它只是一个指向分配到堆内存数据的指针。如果用 `Box` 去封装更多字段，那么还可以让 `Json` 更紧凑。

这里值得注意的是 Rust 实现这个数据结构有多简单。如果是 C++，那你可能要为此写一个类：

```
    class JSON {
    private:
        enum Tag {
            Null, Boolean, Number, String, Array, Object
        };
        union Data {
            bool boolean;
            double number;
            shared_ptr<string> str;
            shared_ptr<vector<JSON>> array;
            shared_ptr<unordered_map<string, JSON>> object;
```

```
        Data() {}
        ~Data() {}
        ...
    };

    Tag tag;
    Data data;

public:
    bool is_null() const { return tag == Null; }
    bool is_boolean() const { return tag == Boolean; }
    bool get_boolean() const {
        assert(is_boolean());
        return data.boolean;
    }
    void set_boolean(bool value) {
        this->~JSON();  // 清理字符串/数组/对象值
        tag = Boolean;
        data.boolean = value;
    }
    ...
};
```

写了 30 行代码，还不能真正开始干活儿：这个类还需要构造函数、析构函数和赋值操作符。另一种思路是使用父子类，比如基类是 JSON，子类是 JSONBoolean、JSONString，等等。无论采用哪种方式，操作完成以后，这个 C++ JSON 类库一定会包含十几个方法。其他程序员要使用它必须先好好看看才行。而整个 Rust 枚举仅用了 8 行代码。

10.1.4　泛型枚举

枚举也可以泛型化。Rust 标准库中最常用的两个枚举类型都是泛型枚举：

```
enum Option<T> {
    None,
    Some(T)
}

enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

我们对这两个类型已经非常熟悉了，而泛型枚举的语法跟泛型结构体一样。有一个小小的细节要注意，如果类型 T 是 Box 或其他智能指针类型，Rust 就会省掉 Option<T> 的标签字段。因此 Option<Box<i32>> 在内存中只用 1 个机器字存储。0 表示 None，非零表示 Some 封装的值。

基于泛型的数据结构可以用寥寥几行代码实现：

```
// T类型值的有序集合
enum BinaryTree<T> {
    Empty,
```

```
        NonEmpty(Box<TreeNode<T>>)
    }

    // BinaryTree的节点
    struct TreeNode<T> {
        element: T,
        left: BinaryTree<T>,
        right: BinaryTree<T>
    }
```

这几行代码定义了一个 `BinaryTree` 类型，其可以存储类型 `T` 任何数量的值。

这两个简洁的定义浓缩了大量信息，因此有必要逐字逐句地解读一下。每个 `BinaryTree` 值要么是 `Empty`，要么是 `NonEmpty`。如果是 `Empty`，那它根本不包含任何数据。如果是 `NonEmpty`，那它包含一个 `Box`，也就是一个指向位于堆内存的 `TreeNode` 的指针。

每个 `TreeNode` 值包含一个实际的元素，以及另外两个 `BinaryTree` 值。这意味着树可以包含子树，因此 `NonEmpty` 树可以包含任意个后代节点。

图 10-3 展示了一个 `BinaryTree<&str>` 值的草图。与 `Option<Box<T>>` 一样，Rust 也省掉了其标签字段，因此一个 `BinaryTree` 值只占 1 个机器字。

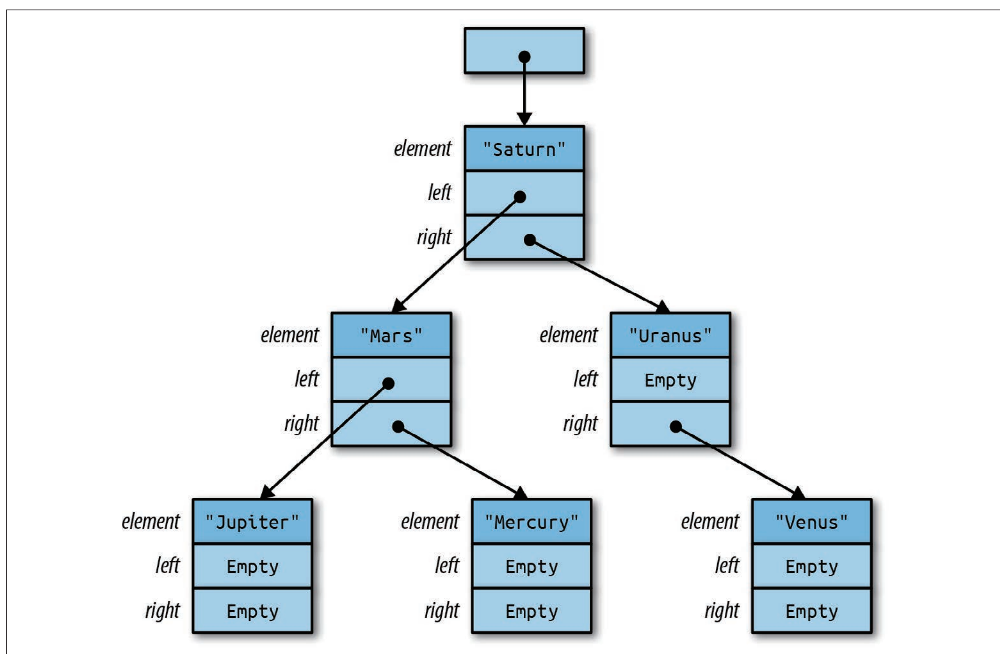


图 10-3：一个包含 6 个字符串的 `BinaryTree`

创建这个树的任何特定节点都很简单：

```
    use self::BinaryTree::*;
    let jupiter_tree = NonEmpty(Box::new(TreeNode {
        element: "Jupiter",
```

```
        left: Empty,
        right: Empty
    }));
```

大一点的树可以基于小一点的树来创建：

```
    let mars_tree = NonEmpty(Box::new(TreeNode {
        element: "Mars",
        left: jupiter_tree,
        right: mercury_tree
    }));
```

自然，这里的赋值会把 jupiter_node 和 mercury_node 的所有权转移给它们的新父节点。

创建树的剩余部分也一样。根节点与其他节点没有区别：

```
    let tree = NonEmpty(Box::new(TreeNode {
        element: "Saturn",
        left: mars_tree,
        right: uranus_tree
    }));
```

本章后面将介绍如何在 BinaryTree 类型上实现一个 add 方法，到时候就可以这样写了：

```
    let mut tree = BinaryTree::Empty;
    for planet in planets {
        tree.add(planet);
    }
```

不管具有什么语言背景，用 Rust 创建类似 BinaryTree 的数据结构可能都需要练习一下。Box 放到哪里在一开始可能没那么明显。像图 10-3 那样画出该数据结构在内存中的布局有助于理解自己的设计是否可行。然后再反过来把图示转换成代码：每组矩形代表一个结构体（或元组），每个箭头代表一个 Box（或其他智能指针）。确定每个字段的类型都要费点脑筋，但也没那么难。而搞清这些问题的回报就是能够掌握自己程序的内存用度。

现在该说一说本章开头所提到的“代价”了。枚举的标签字段要占用一点内存，最坏的情况下可能要 8 字节，但这点内存通常是微不足道的。要说枚举真正的缺点（如果可以算缺点的话），那就是 Rust 代码不能无所顾忌地访问枚举的字段，不管这些字段是否真正在值中存在：

```
    let r = shape.radius;   // 错误：shape类型没有radius字段
```

访问枚举数据的唯一方式是一种安全的方式：使用模式。

10.2　模式

回忆一下本章前面定义的这个 RoughTime 类型：

```
    enum RoughTime {
        InThePast(TimeUnit, u32),
        JustNow,
        InTheFuture(TimeUnit, u32)
    }
```

假设有一个 RoughTime 值，你需要把它显示在网页上。那必须访问这个值中的 TimeUnit 和 u32 字段。在 Rust 中不能通过 rough_time.0 和 rough_time.1 直接访问它们，毕竟 rough_time 的值也可能是 RoughTime::JustNow。那么，怎么把这些数据取出来呢？

要用 match 表达式：

```
 1 fn rough_time_to_english(rt: RoughTime) -> String {
 2     match rt {
 3         RoughTime::InThePast(units, count) =>
 4             format!("{} {} ago", count, units.plural()),
 5         RoughTime::JustNow =>
 6             format!("just now"),
 7         RoughTime::InTheFuture(units, count) =>
 8             format!("{} {} from now", count, units.plural())
 9     }
10 }
```

match 匹配模式。具体到这个例子，**模式**就是第 3、5、7 行 => 符号前面的部分。匹配 RoughTime 值的模式与创建 RoughTime 值使用的表达式看起来几乎一样，这不是巧合：表达式**产出**值，模式**消费**值，两者使用了很多相同的语法。

我们来逐步分析 match 表达式运行时会发生什么。假设 rt 的值是 RoughTime::InTheFuture (TimeUnit::Months, 1)。Rust 先尝试用第 3 行的模式匹配它。如图 10-4 所示，不匹配。

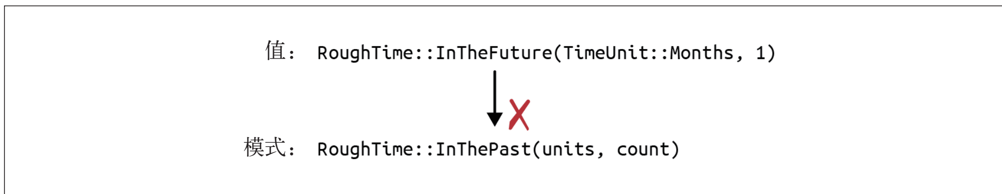


图 10-4：RoughTime 值与模式不匹配

枚举、结构体或元组在匹配模式时，会从左到右对比模式的每个组件，依次检查当前值是否与之匹配。如果不匹配，就前进到下一个模式。

第 3 行和第 5 行的模式都不匹配。但第 7 行的模式匹配成功了，如图 10-5 所示。

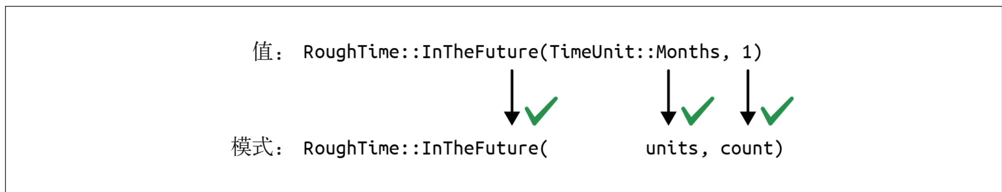


图 10-5：匹配成功

模式中包含的简单标识符，比如 units 和 count，在模式后面的代码中会成为局部变量。值中的内容会被复制或转移到这两个新变量中。在此，Rust 把 TimeUnit::Months 保存到 units 中，把 1 保存到 count 中，随后运行第 8 行代码，返回字符串 "1 months from now"。

这个输出有一个小语法问题：没有区分英文的单复数。为此可以再给 match 添加一个分支：

```
RoughTime::InTheFuture(unit, 1) =>
    format!("a {} from now", unit.singular()),
```

这个分支只在 count 字段恰好等于 1 时匹配，而且新分支必须加到第 7 行以前。如果把它放在末尾，那 Rust 将永远不会运行到它，因为第 7 行的模式匹配 InTheFuture 的所有值。Rust 编译器对于这类错误会给出“unreachable pattern”（执行不到的模式）的警告。

然而，即使是这个新代码，RoughTime::InTheFuture(TimeUnit::Hours, 1) 仍然有一个问题，即结果 "a hour from now" 并不完全正确。这可是一句英语。这个问题可以通过给 match 再增加一个分支来解决。

由这个例子可知，模式与枚举简直是“天作之合”，甚至连枚举包含的数据都能匹配。如此灵活、强大的 match 表达式足以替代 C 的 switch 语句。

到目前为止，本书只介绍了匹配枚举值的模式。事实上模式比这要强大得多。Rust 模式本身就是一门迷你语言，如表 10-1 所示。本章剩下的内容将主要介绍这个表中的特性。

表10-1：模式

模式类型	示　　例	说　　明
字面量	`100` `"name"`	匹配确切的值；const 声明的名字也可以
范围	`0 ... 100` `'a' ... 'k'`	匹配范围中的任意值，包括最终值
通配符	`_`	匹配任意值并忽略该值
变量	`name` `mut count`	类似 _，但会把匹配的值转移或复制到新的局部变量
ref 变量	`ref field` `ref mut field`	不转移或复制匹配的值，而是借用匹配值的引用
子模式绑定	`val @ 0 ... 99` `ref circle @ Shape::Circle { .. }`	匹配 @ 右侧的模式，使用左侧的变量名

枚举模式	`Some(value)`
	`None`
	`Pet::Orca`	

元组模式	`(key, value)` `(r, g, b)`	

结构体模式	`Color(r, g, b)`
	`Point { x, y }`
	`Card { suit: Clubs, rank: n }`
	`Account { id, name, .. }`	

引用	`&value` `&(k, v)`	只匹配引用值
多个模式	`'a' | 'A'`	仅限 match（不能在 let 等中使用）
护具表达式	`x if x * x <= r2`	仅限 match（不能在 let 等中使用）

10.2.1 模式中的字面量、变量和通配符

前面我们看到了如何使用 match 表达式匹配枚举，其他类型也可以匹配。如果想实现 C 的 switch 语句，可以在 match 表达式中使用整数值。0、1 等整数值可以作为模式使用：

```
match meadow.count_rabbits() {
    0 => {}  // 没什么可说的
    1 => println!("A rabbit is nosing around in the clover."),
    n => println!("There are {} rabbits hopping about in the meadow", n)
}
```

模式 0 匹配草地（meadow）上没有兔子。1 匹配只有一只兔子。两只或更多兔子由模式 n 匹配。n 是一个变量名，匹配任意值。匹配的值会被转移或复制到一个新局部变量中。在这里，meadow.count_rabbits() 的值会被保存到新局部变量 n 中，然后打印出来。

其他类型的字面量也可以用作模式，包括布尔值、字符，甚至字符串：

```
let calendar =
    match settings.get_string("calendar") {
        "gregorian" => Calendar::Gregorian,
        "chinese" => Calendar::Chinese,
        "ethiopian" => Calendar::Ethiopian,
        other => return parse_error("calendar", other)
    };
```

在这个例子中，other 作为一个兜底模式，与上个例子中的 n 类似。它们都相当于 switch 语句中的 default 分支，用于匹配不与任何其他模式匹配的值。

如果你需要一个兜底模式，但又不在乎匹配的值，那么可以使用 _ 作为模式，它是一个**通配符**：

```
let caption =
    match photo.tagged_pet() {
        Pet::Tyrannosaur => "RRRAAAAAHHHHHH",
        Pet::Samoyed => "*dog thoughts*",
        _ => "I'm cute, love me" // 通用标题，任何宠物（pet）都适用
    };
```

通配符模式匹配任意值，但不保存匹配的值。由于 Rust 要求每个 match 表达式都要处理所有可能的值，因此通常最后都会有一个通配符。即便你非常确定其他情况不会发生，也必须至少加上一个后备的诧异分支：

```
// 有很多形状（shape），但我们只支持“选择”某些文本，
// 或者一个矩形区域中的所有内容。不能选择椭圆或梯形
match document.selection() {
    Shape::TextSpan(start, end) => paint_text_selection(start, end),
    Shape::Rectangle(rect) => paint_rect_selection(rect),
    _ => panic!("unexpected selection type")
}
```

有一点必须注意，即已有的变量不能用作模式。假设我们要用六边形实现一个棋类游戏，玩家通过点击来走棋。为确认点击有效，可能会这样写代码：

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex> {
    match point_to_hex(click) {
        None =>
            Err("That's not a game space."),
        Some(current_hex) =>  // 想要匹配用户单击了当前的六边形（current_hex）
                              // （但这样不可以：原因见下面）
            Err("You are already there! You must click somewhere else."),
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

但这样写不行，因为模式中的标识符会引入**新**变量。这里的模式 `Some(current_hex)` 会创建新的局部变量 `current_hex`，它会遮住参数 `current_hex`。Rust 会对这行代码发出几个警告，其中最主要的是 `match` 的最后一个分支无法运行到。要解决问题，可以使用一个 `if` 表达式：

```
Some(hex) =>
    if hex == current_hex {
        Err("You are already there! You must click somewhere else")
    } else {
        Ok(hex)
    }
```

稍后还会讨论到护具（guard），也可以用它来解决这个问题。

10.2.2 元组与结构体模式

元组模式匹配元组，适合在一个 `match` 表达式中同时匹配多个数据：

```
fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;
    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "at the origin",
        (_, Equal) => "on the x axis",
        (Equal, _) => "on the y axis",
        (Greater, Greater) => "in the first quadrant",
        (Less, Greater) => "in the second quadrant",
        _ => "somewhere else"
    }
}
```

结构体模式使用花括号，类似结构体表达式，其中每个字段都是一个子模式：

```
match balloon.location {
    Point { x: 0, y: height } =>
        println!("straight up {} meters", height),
    Point { x: x, y: y } =>
        println!("at ({}m, {}m)", x, y)
}
```

在这个例子中，如果第一个分支匹配，那么 `balloon.location.y` 就会被存储在新局部变量 `height` 中。

假设 balloon.location 的值是 Point { x: 30, y: 40 }。那么，与往常一样，Rust 会依次检查每个模式的每个组件，如图 10-6 所示。

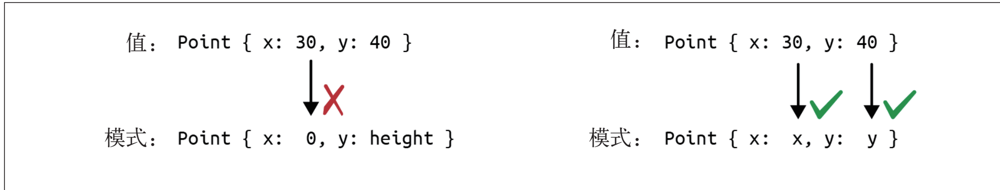


图 10-6：匹配结构体模式

第二个分支匹配，因此会输出“at {30m, 40m}”。

Point { x: x, y: y } 这样的模式在匹配结构体时很常用，但重复的名字看起来有点乱。为此，Rust 允许在这种情况下使用简写形式 Point { x, y }，含义不变。这个模式同样会把一个点的 x 和 y 字段分别存储到新局部变量 x 和 y 中。

对于复杂的结构体，如果只关心其中几个字段，那么就算使用这种简写形式也会显得很啰唆：

```
match get_account(id) {
    ...
    Some(Account {
            name, language, // <--- 只关心这两个字段
            id: _, status: _, address: _, birthday: _, eye_color: _,
            pet: _, security_question: _, hashed_innermost_secret: _,
            is_adamantium_preferred_customer: _ }) =>
        language.show_custom_greeting(name)
}
```

这时候，可以使用 .. 告诉 Rust 你并不关心其他字段：

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name)
```

10.2.3　引用模式

对于引用，Rust 支持两种模式：ref 模式和 & 模式。前者借用匹配值的元素，后者匹配引用。先来看 ref 模式。

匹配不可复制的值会转移值。仍然以前面的 Account 结构体为例，以下代码是无效的：

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account); // 错误：使用转移的值account
    }
}
```

这里，字段 account.name 和 account.language 被转移到了局部变量 name 和 language 中。account 的其他元素则被丢弃了。这就是不能在后面的方法中使用 account 的原因。

如果 name 和 language 都是可以复制的值，Rust 则会复制这两个字段，而不是转移它们。这时候前面的代码是有效的。但假设它们就是不可复制的 String 该怎么办？

我们需要一种模式来**借用**而不转移匹配的值。关键字 ref 就是做这个用的：

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        ui.show_settings(&account); // 没问题
    }
}
```

现在局部变量 name 和 language 中存储的是对 account 中对应字段的引用。由于 account 只是被借用而不是被消费了，因此后面的方法继续使用它是没问题的。

还可以使用 ref mut 借用 mut 引用：

```
match line_result {
    Err(ref err) => log_error(err),  // err是&Error（共享的ref）
    Ok(ref mut line) => {            // line是&mut String（可修改的ref）
        trim_comments(line);         // 就地修改字符串
        handle(line);
    }
}
```

模式 Ok(ref mut line) 匹配任何成功的结果，并借用该结果中存储的值的 mut 引用。

与 ref 模式对应的是 & 模式。以 & 开头的模式匹配引用。

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

在这个例子中，假设 sphere.center() 返回对 sphere 中一个私有字段的引用（这是 Rust 中常见的做法），返回值是一个 Point3d 的地址。如果中心位于原点，那么 sphere.center() 返回 &Point3d { x: 0.0, y: 0.0, z: 0.0 }。

因而模式匹配过程如图 10-7 所示。

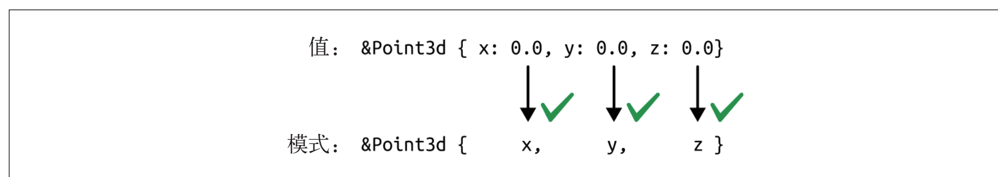


图 10-7：引用的模式匹配

这有点不好理解，因为 Rust 在这里跟进了指针，而这种操作通常要使用 * 而不是 & 操作符。请大家记住：表达式和模式天生是相反的。表达式 (x, y) 用两个值创建一个新元组，模式 (x, y) 则相反：它匹配元组并将其破坏后取出两个值。对 & 而言也一样：表达式中的 & 创建引用，模式中的 & 匹配引用。

匹配引用遵循前面介绍过的所有规则。生命期是必要条件。不能对共享引用采取 mut 操作。不能从引用（包括 mut 引用）中转移出值。在匹配 &Point3d { x, y, z } 时，变量 x、y 和 z 取得的是坐标的副本，原始的 Point3d 值原封未动，因为那些字段是可复制的。对包含不可复制字段的结构体采取同样的做法则会导致错误：

```
match friend.borrow_car() {
    Some(&Car { engine, .. }) =>   // 错误：借用的值不能转移
        ...
    None => {}
}
```

从借来的车上拆零件可不太好，Rust 不会支持这种行为。此时可以使用 ref 模式来借用对零件的引用——不据为己有就行了。

```
Some(&Car { ref engine, .. }) =>   // 可以：engine是一个引用
```

下面再看一个 & 模式的例子。假设我们有一个迭代器 chars，用于迭代字符串中的字符。它有一个方法 chars.peek()，该方法返回 Option<&char>，即对下一个字符的引用（如果有下一个字符的话）。（这种带 peek() 方法的迭代器实际上会返回一个 Option<&ItemType>，第 15 章将介绍。）

程序可以使用 & 模式取得引用所指向的字符：

```
match chars.peek() {
    Some(&c) => println!("coming up: {:?}", c),
    None => println!("end of chars")
}
```

10.2.4　匹配多种可能性

竖线（|）可用于在一个 match 分支中组合多个模式：

```
let at_end =
    match chars.peek() {
        Some(&'\r') | Some(&'\n') | None => true,
        _ => false
    };
```

在表达式中，| 是**按位或**操作符，但在这里它更像正则表达式中的 | 符号。如果 chars.peek() 匹配三个模式中的任何一个，则 at_end 会被设置为 true。

使用 ... 可以匹配某个范围中的值。范围模式包含起点值和终点值，即 '0' ... '9' 匹配所有 ASCII 数字：

```
match next_char {
    '0' ... '9' =>
        self.read_number(),
    'a' ... 'z' | 'A' ... 'Z' =>
        self.read_word(),
    ' ' | '\t' | '\n' =>
        self.skip_whitespace(),
    _ =>
```

```
        self.handle_punctuation()
    }
```

模式中的范围是**全纳**（inclusive）的，因此 `'0'` 和 `'9'` 都匹配 `'0' ... '9'`。相应地，范围表达式（中间是两个点，比如 `for n in 0..100`）则是半开放或**互斥**的（包含 `0` 但不包含 `100`）。之所以存在这种不一致的现象，主要是因为互斥范围对循环和片段更有用，而全纳范围在模式匹配中更有用。

10.2.5　模式护具

使用 `if` 关键字给 `match` 分支添加**护具**。只有在护具求值为 `true` 时匹配才成功：

```
match robot.last_known_location() {
    Some(point) if self.distance_to(point) < 10 =>
        short_distance_strategy(point),
    Some(point) =>
        long_distance_strategy(point),
    None =>
        searching_strategy()
}
```

如果模式转移值，则不能给它添加护具。如果护具求值为 `false`，那 Rust 会继续匹配下一个模式。但如果待匹配的值被转移了，那 Rust 也就没法再继续了。因此前面的代码只对可复制的 `point` 有效，否则就会导致错误：

```
error[E0008]: cannot bind by-move into a pattern guard
  --> enums_move_into_guard.rs:19:18
   |
19 |             Some(point) if self.distance_to(point) < 10 =>
   |                  ^^^^^ moves value into pattern guard
```

解决方法是将模式修改为借用 `point`，而不是转移值：`Some(ref point)`。

10.2.6　@模式

最后，*`x @ pattern`* 匹配给定的 *`pattern`*，但成功之后，不是基于匹配值的元素来创建变量，而是把匹配值整个转移或复制到一个变量 `x` 中。比如，假设有如下代码：

```
match self.get_selection() {
    Shape::Rect(top_left, bottom_right) =>
        optimized_paint(&Shape::Rect(top_left, bottom_right)),
    other_shape =>
        paint_outline(other_shape.get_outline()),
}
```

注意，第一个分支拆解出 `Shape::Rect` 值，仅仅是为了在下一行再重新创建一个相同的 `Shape::Rect` 值。这里可以用 `@` 模式重写为：

```
rect @ Shape::Rect(..) =>
    optimized_paint(&rect),
```

`@` 模式在匹配范围时也有用：

```
match chars.next() {
    Some(digit @ '0' ... '9') => read_number(digit, chars),
    ...
}
```

10.2.7　在哪里使用模式

除了最明显地用在 `match` 表达式中，模式还可以用在其他一些地方，通常用于代替标识符。无论用在哪里，用途都不变：通过模式匹配实现拆解值，而不是仅仅把值保存在一个变量中。

这意味着模式可用于……

```
// ……将结构体拆解为3个新的局部变量
let Track { album, track_number, title, .. } = song;

// ……拆解作为函数参数的元组
fn distance_to((x, y): (f64, f64)) -> f64 { ... }

// ……迭代HashMap的键和值
for (id, document) in &cache_map {
    println!("Document #{}: {}", id, document.title);
}

// ……自动对传给闭包的参数解引用（有时候你想要副本，
// 而其他代码传的是引用，这时候就方便了）
let sum = numbers.fold(0, |a, &num| a + num);
```

前面每个例子都能节省 2 到 3 行样板代码。其他语言也有同样的概念，JavaScript 中叫**解构**（destructuring），Python 中叫**解包**（unpacking）。

注意，这 4 个例子中使用的都是保证能匹配的模式。模式 `Track { album, track_number, title, .. }` 匹配结构体 `Track` 中指定的值；`(x, y)` 匹配任意 `(f64, f64)` 元组，等等。始终都可以匹配的模式在 Rust 中是一种特殊模式，叫**不可驳模式**（irrefutable pattern）。它们是仅有的可以出现以上代码中所示 4 个位置（`let` 后面、函数参数中、`for` 后面、闭包参数中）的模式。

可驳模式（refutable pattern）指的是那些可能不会匹配的模式，比如 `Ok(x)` 不会匹配错误结果，`'0' ... '9'` 则不会匹配字符 `'Q'`。可驳模式可用于 `match` 表达式中，因为 `match` 表达式是专门为其设计的：如果一个模式匹配失败，那么还有下一个。前面 4 个例子中所示的是 Rust 程序中可以利用模式之便的位置，但语言在此时不允许模式失败。

可驳模式还可以用在 `if let` 和 `while let` 表达式中，用来……

```
// ……只处理一种特定的枚举变体
if let RoughTime::InTheFuture(_, _) = user.date_of_birth() {
    user.set_time_traveler(true);
}

// ……只在查表成功时运行某些代码
if let Some(document) = cache_map.get(&id) {
```

```
        return send_cached_response(document);
    }

    // ……不成功则重复做一些事
    while let Err(err) = present_cheesy_anti_robot_task() {
        log_robot_attempt(err);
        // 让用户再次尝试（可能还是人类）
    }

    // ……手工遍历一个迭代器
    while let Some(_) = lines.peek() {
        read_paragraph(&mut lines);
    }
```

关于这两个表达式的详细介绍，参见 6.4 节和 6.5 节。

10.2.8　填充二叉树

本章前面承诺过要展示如何实现 `BinaryTree::add()` 方法，该方法用于向 `BinaryTree` 中添加相同类型的子节点：

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}
```

与实现这个方法相关的模式前面都介绍过了。本书不会介绍二叉树，但熟悉这个概念的读者应该可以理解 Rust 是如何实现这种数据结构的：

```
 1 impl<T: Ord> BinaryTree<T> {
 2     fn add(&mut self, value: T) {
 3         match *self {
 4             BinaryTree::Empty =>
 5                 *self = BinaryTree::NonEmpty(Box::new(TreeNode {
 6                     element: value,
 7                     left: BinaryTree::Empty,
 8                     right: BinaryTree::Empty
 9                 })),
10             BinaryTree::NonEmpty(ref mut node) =>
11                 if value <= node.element {
12                     node.left.add(value);
13                 } else {
14                     node.right.add(value);
15                 }
16         }
17     }
18 }
```

第 1 行告诉 Rust 我们要在有序类型的 `BinaryTree` 上定义一个方法。这里使用了与 9.5 节讲过的在泛型结构体上定义方法一样的语法。

如果当前的树 `*self` 是空的，那就简单了，第 5~9 行运行，把 `Empty` 的树改成 `NonEmpty` 的树。这里调用 `Box::new()` 在堆上分配了一个新的 `TreeNode`。运行之后，这个树包含一个元素，其左、右子树都是 `Empty` 的。

如果 `*self` 不是空的，那么第 10 行的模式匹配成功：

```
BinaryTree::NonEmpty(ref mut node) =>
```

这个模式从 `Box<TreeNode<T>>` 中借用了一个可修改引用，因此我们可以访问并修改该树节点中的数据。这个引用叫 `node`，位于第 11~15 行的作用域中。由于这个节点中已经有一个元素了，因此代码必须递归调用 `.add()` 将新元素添加到其左或右子树上。

可以像下面这样调用这个新方法：

```
let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...
```

10.3 设计的考量

Rust 的枚举对系统编程来讲可能是个新东西，但它不是一个新想法。实际上它们在函数式编程语言中已经存在 40 多年了，而且有着各种学术气息浓厚的名字，比如**代数数据类型**（algebraic data type）。不清楚为什么 C 流派其他语言支持这个概念的那么少。有可能是因为对编程语言设计者来说，要做到变体（variant）、引用、可变性（mutability）和内存安全 4 方面兼得，确实是极大的挑战。函数编程语言不需要可变性，因为没必要。而 C 的 `union` 同时支持变体、指针和可变性，但其骇人的不安全性也导致人们不到万不得已不会考虑它。Rust 的借用检查机制魔法般地做到了既把这 4 方面组合起来，又没有在任何一方面打折扣。

编程就是处理数据。简短、快捷、优雅的程序和庞大、缓慢、充斥着连接绑定及虚拟方法调用的杂混代码之间的区别，就在于数据形态是否合理。

这正是枚举的用武之地。换句话说，枚举其实是特定数据形态的设计工具。一个值可能是一个值，也可能是另外一个值，还有可能什么也不是。这种情况很普遍，而相较于叠床架屋式的类层级，无论从哪个维度看枚举都更胜一筹：速度更快、更加安全、代码更少，写文档也更简单。

限制因素是灵活性。枚举的最终用户不能扩展枚举，比如添加一个新变体。要添加变体只能修改枚举的声明。而一旦修改了声明，现有代码就不能用了。必须重新检查匹配枚举中每个变体的每个 `match` 表达式，因为需要给它们都添加一个新分支以处理新变体。有时候，牺牲灵活性换取简单恰恰是合理的。有谁认为 JSON 的数据结构会变来变去呢？而某些情况下，在枚举改变时重新检查所有用例实际上正是我们希望的。比如，某个编译器使用 `enum` 表示一种编程语言的不同操作符。新增一个操作符，**必然要求**重新检查处理操作符的所有代码。

不过有时候还是需要更多灵活性的。Rust 为此提供了特型，这也是下一章的主题。

第 11 章

特型与泛型

> 计算机科学家倾向于处理不统一的结构，情形 1、情形 2、情形 3……而数学家倾向于拥有一个适用于整个系统的统一的公理。
>
> ——Donald Knuth

编程中的一个重大发现，就是可以编写对很多不同类型的值进行操作的代码，**甚至是那些尚未发明的类型**。下面是两个例子。

- `Vec<T>` 是泛型：你可以创建一个包含任何类型值的向量，包括在自己程序里面定义的、`Vec` 作者决不会想到的类型。
- 很多东西有 `.write()` 方法，包括 `File` 和 `TcpStream`。代码可以通过引用获得一个书写器（writer），不管是什么书写器，然后将数据发送给它。此时不必关心书写器的类型。将来，如果有人写了一个新的书写器类型，那你的代码照样支持它。

当然，这种能力对 Rust 而言不是什么新东西。这叫作**多态性**，是 20 世纪 70 年代在编程语言领域非常火的一种技术。到了今天，多态性已经无所不在了。Rust 对多态性的支持构建于两个相关特性之上：特型（trait）和泛型（generic）。很多程序员对这两个概念不陌生，但 Rust 在 Haskell 类型类（typeclass）的启发下采用了一种新思路。

特型是 Rust 对接口或抽象基类的实现。首先，它看起来就像 Java 或 C# 中的接口。比如写字节的特型叫 `std::io::Write`，它在标准库中定义的开头是这样的：

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;
    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

这个特型声明了不少方法，这里仅展示了 3 个。

标准类型 File 和 TcpStream 都实现了 std::io::Write，Vec<u8> 也实现 std::io::Write。那么这 3 个类型就都有了名为 .write()、.flush() 等的方法。使用书写器但又不关心其类型的代码如下：

```
use std::io::Write;

fn say_hello(out: &mut Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

out 的类型是 &mut Write，含义是“一个对实现 Write 特型的任意值的可修改引用”。

```
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?; // 没问题

let mut bytes = vec![];
say_hello(&mut bytes)?; // 同样没问题
assert_eq!(bytes, b"hello world\n");
```

本章将首先介绍如何使用特型、它们的工作原理，以及如何自定义特型。到目前为止，我们了解到的特型只是冰山一角。可以通过它给现有类型甚至 str 和 bool 等内置类型添加扩展方法。本章接下来解释为什么把特型添加给类型不用花额外内存，以及如何在没有虚拟方法调用开销的前提下使用特型。我们会看到，内置特型其实是 Rust 语言为操作符重载及其他特性提供的钩子（hook）。本章还会介绍 Self 类型、关联方法和关联类型这 3 个 Rust 从 Haskell 中抄来的特性，这些特性可以优雅地解决别的语言只能以权宜之计和黑招数（hack）解决的问题。

泛型是 Rust 中另一种多态性的实现。类似 C++ 模板，泛型函数或类型可以搭配很多种不同类型的值使用。

```
/// 给定两个值，取出较小的
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

这个函数中的 <T: Ord> 代表 min 可以使用实现 Ord 特型的任意类型参数 T（也就是任意有序类型）。编译器会为每个实际使用的类型 T 生成定制的机器码。

泛型与特型紧密相关。Rust 让我们在使用 <= 操作符比较类型 T 的两个值之前，先行声明 T: Ord 这个条件（叫作**绑定**）。本章接下来会谈谈 &mut Write 和 <T: Write> 为什么相似、它们有哪些区别，以及它们各自适用的情形。

11.1　使用特型

特型是一种任何类型都可以选择支持或不支持的特性。通常，特型代表一种能力，即某类

型能做什么。

- 实现 `std::io::Write` 的值可以执行写字节操作。
- 实现 `std::iter::Iterator` 的值可以产生值的序列。
- 实现 `std::clone::Clone` 的值可以在内存中克隆自身。
- 实现 `std::fmt::Debug` 的值可以使用 `println!()` 的 `{:?}` 格式说明符打印出来。

以上列举的几个特型都是 Rust 标准库的一部分，很多标准库类型实现了它们。

- `std::fs::File` 实现了 `Write` 特型，从而能够将字节写入本地文件。`std::net::TcpStream` 则是将字节写入网络连接。`Vec<u8>` 同样实现了 `Write`。在字节向量上每次调用 `.write()`，都可以在向量末尾追加一些数据。
- `Range<i32>`（即 `0..10` 的类型）实现了 `Iterator` 特型，而与切片、散列表等相关的其他迭代器类型也实现了它。
- 大多数标准库类型实现了 `Clone`。例外情况主要是 `TcpStream` 这种不仅仅表示内存中数据的类型。
- 类似地，大多数标准库类型支持 `Debug`。

提到使用特型方法，有一个不同寻常的规则：特型本身必须在作用域中。否则，特型的所有方法都是隐藏的。

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // 错误：没有找到名为write_all的方法
```

此时，编译器会打印一条友好的错误消息，建议你添加 `use std::io::Write;`。还确实管用：

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // 可以
```

Rust 设定这个规则是因为（正如本章后面会讲到的）你的代码可以使用特型给任何类型添加新方法，即使是 `u32` 和 `str` 这种标准库类型也不例外。第三方包也可以做同样的事。显然，这可能会导致命名冲突。但是，因为 Rust 要求必须导入自己想用的特型，所以第三方包可以自由利用这个超能力，而在实践中冲突并不多见。

`Clone` 和 `Iterator` 方法之所以在没有特定导入的情况下也能使用，是因为它们默认一直在作用域中。它们是标准前置模块的一部分，所以 Rust 会将其自动导入到所有模块中。事实上，前置模块很大程度上可以说是一个精心挑选的特型集合。第 13 章会介绍其中很多特型。

C++ 和 C# 程序员可能已经注意到了，特型方法很像虚拟方法。同样，类似上面的调用非常快，跟任何其他方法调用一样快。简单地说，就是没有发生多态。很明显，`buf` 是一个向量而非文件或网络连接。编译器可以发出对 `Vec<u8>::write()` 的简单调用，甚至可以将该方法行内化。（C++ 和 C# 的思路也差不多，只是子类化有可能会阻止这么做。）只有通过 `&mut Write` 调用才会产生虚拟方法调用的消耗。

11.1.1 特型目标

在 Rust 中有两种方式使用特型编写多态化代码：特型目标和泛型。接下来先看看如何使用

特型目标，然后再说泛型。

Rust 不允许声明 `Write` 类型的变量：

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: Write = buf; // 错误：Write没有固定大小
```

变量的大小必须在编译时就知道，而实现 `Write` 的类型可以是任意大小。

熟悉 C# 和 Java 的程序员可能觉得不可思议，但原因很简单。在 Java 中，`OutputStream`（与 `std::io::Write` 类似的 Java 标准接口）类型的变量是一个引用，指向任何实现 `OutputStream` 的对象。事实上，这个变量是一个引用不言而喻。对 C# 以及其他大多数语言中的接口来说，也是一样的。

在 Rust 中我们想要的其实也一样，只不过 Rust 中的引用是显式的：

```
let mut buf: Vec<u8> = vec![];
let writer: &mut Write = &mut buf; // 可以
```

指向一个特型类型（如 `writer`）的引用，称为**特型目标**（trait object）。与其他引用类似，特型目标也指向某个值，有生命期，可以是 `mut` 或共享引用。

特型目标的不同之处在于，Rust 在编译时通常不知道引用目标的类型。因此特型目标还要包含一些关于引用目标类型的额外信息。这个信息严格限于 Rust 内部自己使用。在调用 `writer.write(data)` 时，Rust 需要知道类型信息，才能根据 `*writer` 的类型动态调用正确的 `write` 方法。任何人都无法直接查询这个类型信息，Rust 也不支持特型目标 `&mut Write` 到具体类型如 `Vec<u8>` 的向下转型。

11.1.2　特型目标布局

在内存中，特型目标是一个胖指针，包含指向值的指针和指向表示该值类型的表的指针。因此，每个特型目标都占用两个机器字，如图 11-1 所示。

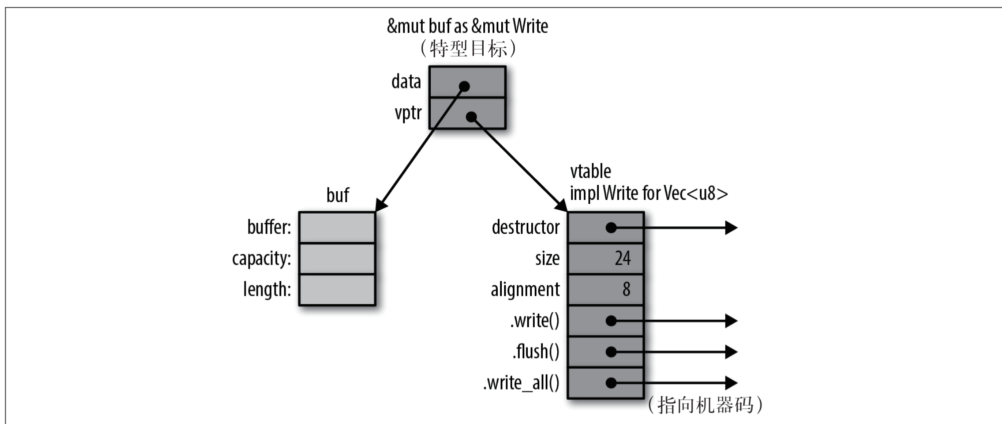


图 11-1：特型目标的内存布局

C++ 同样也有这种运行时类型信息，其被称为**虚拟表**（virtual table）或 vtable。跟 C++ 一样，在 Rust 中，虚拟表同样只在编译时生成一次，由同类型的所有对象共享。图 11-1 中的深灰色表（包括 vtable）都是 Rust 的私有实现细节。再说一次，这些字段和数据结构无法直接访问。相反，当你调用一个特型目标方法时，语言会自动使用虚拟表，以确定调用哪个实现。

经验丰富的 C++ 程序员会注意到 Rust 和 C++ 在内存使用上有一点不同。在 C++ 中，虚拟表指针或 vptr 被保存为结构体的一部分。而 Rust 使用的是胖指针。结构体本身除了自己的字段之外没有别的。这样，一个结构体能够实现很多特型而不必包含同样多的 vptr。即使像 `i32` 这样没有足够空间容纳 vptr 的类型都可以实现特型。

Rust 在必要时会自动将普通引用转换为特型目标。这也是可以在如下示例中将 `&mut local_file` 传给 `say_hello` 的原因：

```
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;
```

`&mut local_file` 的类型是 `&mut File`，而 `say_hello` 的参数类型是 `&mut Write`。因为 `File` 也是一种书写器，所以 Rust 允许这样传递，自动将普通引用转换为特型目标。

类似地，Rust 会很高兴地将 `Box<File>` 转换为 `Box<Write>`（拥有分配在堆内存上的一个书写器的值）：

```
let w: Box<Write> = Box::new(local_file);
```

与 `&mut Write` 类似，`Box<Write>` 也是一个胖指针，包含书写器自身的地址和虚拟表的地址。至于其他指针类型，比如 `Rc<Write>`，也是一样的。

这种转换是创建特型目标的唯一方式。计算机在这里做的事情其实很简单。在转换发生时，Rust 知道引用目标的真正类型（在这里是 `File`），于是它只要给普通指针加上对应虚拟表的地址就把它变成了胖指针。

11.1.3　泛型函数

本章开始时展示了一个 `say_hello()` 函数，该函数接收一个特型目标作为参数。现在把它重写为一个泛型函数：

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

只有类型签名发生了变化：

```
fn say_hello(out: &mut Write)        // 普通函数

fn say_hello<W: Write>(out: &mut W)  // 泛型函数
```

这里让函数变成泛型的是 `<W: Write>`，可以称之为**类型参数**（type parameter）。有了这个类型参数，就意味着在整个函数体内，`W` 始终表示实现了 `Write` 特型的某种类型。按照惯

例，类型参数通常是一个大写字母。

W 到底表示哪种类型取决于如何使用泛型函数：

```
say_hello(&mut local_file)?; // 调用say_hello::<File>
say_hello(&mut bytes)?;      // 调用say_hello::<Vec<u8>>
```

如果把 `&mut local_file` 传给泛型函数 `say_hello()`，那么调用的就是 `say_hello::<File>()`。Rust 会为这个函数生成调用 `File::write_all()` 和 `File::flush()` 的机器码。如果传递的是 `&mut bytes`，那么调用的就是 `say_hello::<Vec<u8>>()`。Rust 同样会为这个版本的函数生成单独的机器码，调用 `Vec<u8>` 上对应的方法。两种情况下，Rust 都会根据参数的类型来推断 W 的类型。当然，可以把类型参数直接写出来：

```
say_hello::<File>(&mut local_file)?;
```

但很少有这样写的必要，因为 Rust 能根据参数推断出类型参数。在此，`say_hello` 泛型函数想要一个 `&mut W` 参数，而传给它的是 `&mut File`，于是 Rust 推断 `W = File`。

如果调用的泛型函数本身没有参数可以提供有用的提示，那恐怕只能明确地把类型参数写出来了：

```
// 调用一个不接收参数的泛型方法collect<C>()
let v1 = (0 .. 1000).collect(); // 错误：无法推断类型
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // 可以
```

有时候，我们需要类型参数提供多种能力。比如，假设我们想打印出向量中最常见的 10 个值，那就需要这些值是可以打印的：

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

但这样还不够。你打算怎样确定哪些值是最常见的？通常的做法是使用值作为散列表的键。这意味着这些值需要支持 `Hash` 和 `Eq` 操作。T 的绑定中必须还得包含 `Debug`。此时对应的语法是使用 + 号：

```
fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

有些类型实现了 `Debug`，有些类型实现了 `Hash`，有些类型支持 `Eq`，还有少数像 `u32` 和 `String` 这样的类型同时实现了这 3 个，如图 11-2 所示。

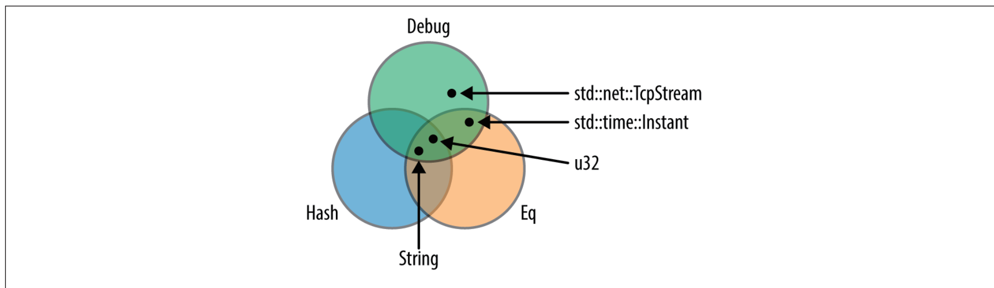


图 11-2：按支持的特型对类型分类

当然，类型参数也可能什么都不绑定。问题是如果不指定任何绑定，那几乎对值什么也做不了。可以转移它，可以把它装箱或放到向量中，也就这样了。

泛型函数可以有多个类型参数：

```
/// 对大型、分段数据集运行查询，参见Google上的MapReduce算法论文
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

如上面的例子所示，绑定会变得很长，因此对眼睛非常不利。Rust 为此提供了使用关键字 `where` 的替代语法：

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

类型参数 `M` 和 `R` 仍然是提前声明的，只不过绑定转移到了另外两行上。这种 `where` 子句也适用于泛型结构体、枚举、类型别名和方法，总之任何允许绑定的地方。

当然，对 `where` 子句的替代方案就是保持简单：找一种不那么集中使用泛型的编程方式。

5.2.2 节介绍了生命期参数的语法。泛型函数可以同时拥有生命期参数和类型参数。生命期参数在前面。

```
/// 返回距离target点最近的candidates点中的引用
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P: MeasureDistance
{
    ...
}
```

这个函数也有两个参数：`target` 和 `candidates`，它们都是引用。这两个参数分别有自己的生命期：`'t` 和 `'c`（正如 5.2.6 节中所讨论的那样）。此外，这个函数可用于任何实现了 `MeasureDistance` 特型的类型 `P`。所以在一个程序中可以对 `Point2d` 值使用它，而在另一个程序中对 `Point3d` 值使用它。

生命期对机器码不会有任何影响。对 `nearest()` 的两次调用使用了相同的类型 `P`，不同的生命期最终调用的都是同一个编译后的函数。只有不同的类型才会让 Rust 编译出同一泛型函数的不同副本。

当然，函数并不是 Rust 中唯一涉及泛型的代码。

- 9.6 节和 10.1.4 节已经介绍过泛型结构体和泛型枚举。
- 个别的方法可以是泛型的，无论定义该方法的类型是不是泛型的。

```
impl PancakeStack {
    fn push<T: Topping>(&mut self, goop: T) -> PancakeResult<()> {
        ...
    }
}
```

- 类型别名也可以是泛型的。

```
type PancakeResult<T> = Result<T, PancakeError>;
```

- 本章后面还会介绍泛型特型。

本节介绍的所有特性，包括绑定、`where` 子句、生命期参数等，可用于所有泛型特性项，而不仅仅是函数。

11.1.4 使用哪一个

有时候到底该使用特型目标还是泛型代码并不是那么显而易见。因为这两者都以特型为基础，所以它们有很多共性。

如果你需要一个混合类型的值全都在一起的集合，那特型目标就是正确的选择。从技术上讲，做一盘泛型沙拉（salad）是完全可能的：

```
trait Vegetable {
    ...
}

struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

但这种设计过于严格了。你看，每份这种沙拉都只能有一种蔬菜，并非所有人都适合。本书的一位作者曾经花 14 美元买了一盘 `Salad<IcebergLettuce>`，至今他都无法从被坑的阴影中走出来。

那怎么做出更好的沙拉呢？考虑到 `Vegetable` 值的大小可能差别很大，因此不能向 Rust 索要 `Vec<Vegetable>`：

```
struct Salad {
    veggies: Vec<Vegetable>  // error: `Vegetable` does not have
                             //        a constant size
}
```

特型目标可以解决这个问题：

```
struct Salad {
    veggies: Vec<Box<Vegetable>>
}
```

每个 `Box<Vegetable>` 可以拥有任意类型的蔬菜（vegetable），而这个“箱子”本身的大小是固定的：两个指针，适合保存在向量中。除了把让人吃的沙拉跟“箱子”放到一起不搭之外，这种结构精确地匹配了我们的要求。不仅是蔬菜，它也适合绘图应用中的形状、游戏中的怪物、网络路由器中可插拔的路由算法，等等。

选择使用特型目标的另一个原因可能是想减少编译后的总代码量。对一个泛型函数，Rust 可能会编译很多次，每次生成一种要使用类型的函数。这样得到的二进制文件会比较大，也就是 C++ 社区中所谓的**代码膨胀**（code bloat）现象。如今，内存已经不是稀缺资源，我们大多数人可以忽略代码大小。然而，受限环境依旧存在。

在涉及制作沙拉和微控制器的场景之外，泛型与特型目标相比有两个重要优势。这就导致泛型在 Rust 中成为更常见的选择。

第一个优势是速度。每次 Rust 编译器为泛型函数生成机器码时，它都知道自己要操作的是什么类型，也知道当时要调用哪个 `write` 方法。这就消除了动态查找的时间。

本章开头示例的泛型函数 `min()` 运行起来就像调用我们单独写的 `min_u8`、`min_i64`、`min_string` 之类的函数一样快。编译器会把它们行内化，就跟其他函数编译后的结果一样。所以在发布构建中，对 `min::<i32>` 的调用可能只是两三个指令。使用常量参数（如 `min(5, 3)`）的调用甚至会更快，因为 Rust 在编译时就会计算出其结果，完全消除了运行时成本。

再看一个调用泛型函数的例子：

```
let mut sink = std::io::sink();
say_hello(&mut sink)?;
```

这里的 `std::io::sink()` 返回一个 `Sink` 类型的书写器，这个书写器会悄无声息地丢弃写入它的所有字节。

Rust 在为这个调用生成机器码时，可以生成调用 `Sink::write_all` 的代码，检查错误，然后再调用 `Sink::flush`。这些都是这个泛型函数体内的代码说要做的。

不过，Rust 通过分析这几个方法也能知道：

- `Sink::write_all()` 什么也不做；
- `Sink::flush()` 什么也不做；
- 两个方法都不会返回错误。

简言之，Rust 拥有完全优化这个函数的全部信息。

再对比一下使用特型目标的行为。Rust 在编译时不可能知道特型目标指向的值的真正类型，只能在运行时确定。因此即使明确传入 `Sink`，也无法消除调用虚拟方法和检查错误的成本。

泛型的第二个优势在于并非所有特型都支持特型目标。特型支持的某些特性，比如静态方法，只对泛型有效，完全没有考虑特型对象。本书后面在遇到这些特性时会明确指出来。

11.2　定义和实现特型

定义特型很简单，只要给它命名并列出特型方法的类型签名即可。假设我们在写一个游戏，可能会定义如下特型：

```
/// 为角色、物品和布景，总之游戏世界中可以在屏幕上看到
/// 的一切定义一个特型
trait Visible {
    /// 在给定画面上渲染对象
    fn draw(&self, canvas: &mut Canvas);

    /// 如果在坐标(x, y)上单击能选择当前对象，就返回true
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

要实现这个特型，使用语法 impl *TraitName* for *Type*：

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
            canvas.write_at(self.x, y, '|');
        }
        canvas.write_at(self.x, self.y, 'M');
    }

    fn hit_test(&self, x: i32, y: i32) -> bool {
        self.x == x
        && self.y - self.height - 1 <= y
        && y <= self.y
    }
}
```

注意，这个 impl 包含对 Visible 特型每个方法的实现，除此之外什么也没有。特型 impl 中定义的一切都必须是特型的特性。如果想添加一个支持 Broom::draw() 的辅助方法，那就需要在单独的 impl 块中定义它：

```
impl Broom {
    /// 由下面的Broom::draw()使用的辅助函数
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}

impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.broomstick_range() {
            ...
        }
        ...
    }
    ...
}
```

11.2.1 默认方法

前面讨论的 Sink 书写器类型可以用几行代码实现。首先，要定义这个类型：

```
/// 忽略任何写入数据的书写器
pub struct Sink;
```

Sink 是一个空结构体，因为不需要它存储任何数据。接下来，为 Sink 提供一个 Write 特型的实现：

```
use std::io::{Write, Result};

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        // 声明已经成功写完整个缓冲
        Ok(buf.len())
```

```
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

现在看来，这很像 `Visible` 特型。但我们也看到了，`Write` 特型有一个 `write_all` 方法：

```
out.write_all(b"hello world\n")?;
```

为什么 Rust 允许 `impl Write for Sink` 不定义这个方法？答案是标准库中 `Write` 特型的定义包含一个 `write_all` 的**默认实现**：

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> {
        let mut bytes_written = 0;
        while bytes_written < buf.len() {
            bytes_written += self.write(&buf[bytes_written..])?;
        }
        Ok(())
    }

    ...
}
```

`write` 和 `flush` 方法是每个书写器必须实现的基本方法。书写器也可以实现 `write_all`，但如果没有，就会使用上面展示的默认实现。

同样，自定义特型也可以使用上面的语法包含默认的方法实现。

标准库中使用默认方法最多的是 `Iterator` 特型，它有一个必需方法（`.next()`）和很多默认方法。第 15 章将解释这是为什么。

11.2.2　特型与其他人的类型

Rust 允许在任意类型上实现任意特型，只要当前包中导入了相关特型或类型即可。

这意味着，每当想给任何类型添加方法时，都可以使用一个特型来完成：

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// 为内置字符类型实现IsEmoji
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

跟其他特型方法一样，这个新的 is_emoji 方法只有 IsEmoji 在当前作用域时才可见。

上面这个特定特型存在的唯一目的是给一个已有的类型 char 添加方法。这种特型称为**扩展特型**。当然，通过编写 impl IsEmoji for str { ... } 等也可以让其他类型实现这个特型。

甚至，可以使用泛型 impl 块让一个类型家族一次性全部实现一个扩展特型。以下扩展特型为所有 Rust 书写器都添加了一个方法：

```
use std::io::{self, Write};

/// 为可以发送HTML的值定义扩展特型
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}

/// 这样就可以向任意std::io::writer写入HTML了
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}
```

impl<W: Write> WriteHtml for W 的意思是："对每个实现了 Write 的类型 W，在这里为 W 再实现特型 WriteHtml。"

要了解在标准类型上实现用户定义特型的价值，可以参考 serde 库。这个库是用于实现序列化的，也就是说，可以使用 serde 把 Rust 数据结构写到磁盘上，以便将来重新加载。这个库定义了一个叫 Serialize 的特型，并在它支持的所有数据类型上都实现了该特型。因此，在 serde 源代码中，有一些代码通过所有像 Vec 和 HashMap 这样标准的数据结构为 bool、i8、i16、i32、数组和元组类型等实现了 Serialize。

最终结果是 serde 为所有这些类型都添加了一个 .serialize() 方法，可以这样使用：

```
use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()>
{
    // 创建一个JSON序列化处理程序将数据写入文件
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);

    // 剩下的事都交给serde的.serialize()方法处理
    config.serialize(&mut serializer)?;
    Ok(())
}
```

前面说过，在实现特型时，相关的特型或类型必须有一个在当前包中是新的。这叫**连贯规则**（coherence rule）。这个规则有助于 Rust 确保特型实现的唯一性。你的代码中不用写 impl Write for u8，就是因为 Write 和 u8 都在标准库中定义了。如果 Rust 包都这样写，

那不同的包很可能会定义 u8 对 Write 的不同实现。当同时使用这些包时，对于给定的方法调用，Rust 也没办法判断到底该使用哪个实现。

（C++ 有一个类似的唯一性限制，叫“一次定义规则”。在典型的 C++ 代码中，这个规则并不由编译器强制遵守，因此除了一些最简单的情形，否则破坏这个规则就会导致未定义行为。）

11.2.3　特型中的Self

特型可以使用关键字 Self 作为类型。比如，标准的 Clone 特型大致是这样定义的（稍有简化）：

```
pub trait Clone {
    fn clone(&self) -> Self;
    ...
}
```

这里用 Self 作为返回类型意味着 x.clone() 的类型就是 x 的类型，不管具体什么类型。如果 x 是 String，那 x.clone() 的类型也是 String，而不是 Clone 或其他实现 Clone 的类型。

类似地，如果像下面这样定义了一个特型：

```
pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}
```

而它有两个实现：

```
impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}
```

那么在第一个 impl 块中，Self 其实是 CherryTree 的别名，而在第二个 impl 块中，Self 是 Mammoth 的别名。这意味着可以把两棵樱桃树（cherry tree）嫁接在一起，也可以让两头猛犸象（mammoth）喜结连理。但是不能把樱桃树和猛犸象弄到一块去。换句话说，self 和 other 的类型必须完全一样。

使用 Self 类型的特型与特型目标不能共存：

```
// 错误：特型Spliceable不能成为引用目标
fn splice_anything(left: &Spliceable, right: &Spliceable) {
    let combo = left.splice(right);
    ...
}
```

随着越来越多地使用特型的高级特性，就能越来越清楚这是为什么。Rust 拒绝这种代码，因为它没办法对 `left.splice(right)` 调用做类型检查。特型目标的核心在于类型到运行时才能知道。而在编译时，Rust 没办法判断 `left` 和 `right` 是不是同一种类型。

特型目标实际上针对的是最简单的特型，这种特型可以使用 Java 中的接口或者 C++ 中的抽象基类实现。特型的更高级的特性很有用，但不能与特型目标共存，因为在它们的特型目标中没有 Rust 做类型检查所需的类型信息。

好了，如果就想实现基因上不可能人工进化，那么可以这样设计一个目标友好的特型：

```
pub trait MegaSpliceable {
    fn splice(&self, other: &MegaSpliceable) -> Box<MegaSpliceable>;
}
```

这个特型与特型目标兼容，因为调用 `.splice()` 方法时，`other` 参数的类型不必跟 `self` 的类型完全一样，只要两者的类型都是 `MegaSpliceable` 就可以通过类型检查。

11.2.4 子特型

可以将一个特型声明为另一个特型的扩展：

```
/// 游戏世界中的角色，可以是玩家或者其他什么小精灵、滴水兽、
/// 小松鼠、食人魔，等等
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

这里的 `trait Creature: Visible` 意味着所有角色都是可见的。为此，所有实现 `Creature` 的类型，必须也实现 `Visible` 特型：

```
impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}
```

为一个类型实现这两个特型的顺序并不重要，但只实现 `Creature` 不实现 `Visible` 是错误的。

子特型类似 Java 或 C# 中的子接口，是一个特型要用另外几个方法扩展已有特型的表达方式。对于上面这个例子而言，所有与角色（`Creature`）有关的代码，也可以使用来自 `Visible` 特型的方法。

11.2.5 静态方法

在大多数面向对象语言中，接口不能包含静态方法或构造函数。然而，Rust 特型可以包含静态方法和构造函数，语法如下：

```
trait StringSet {
    /// 返回一个新的空集合
    fn new() -> Self;

    /// 返回一个包含strings中所有字符串的集合
    fn from_slice(strings: &[&str]) -> Self;

    /// 确定当前集合是否包含特定的value
    fn contains(&self, string: &str) -> bool;

    /// 向当前集合中添加一个字符串
    fn add(&mut self, string: &str);
}
```

所有实现 `StringSet` 特型的类型都必须实现这 4 个关联函数，其中，前两个函数 `new()` 和 `from_slice()` 不接收 `self` 参数。它们充当构造函数。

在非泛型代码中，这些函数可以使用 `::` 语法调用，就跟调用其他静态方法一样：

```
// 创建两个实现了StringSet的假设类型的集合：
let set1 = SortedStringSet::new();
let set2 = HashedStringSet::new();
```

在泛型代码中，其实也差不多，区别在于类型通常是可变的，如下面对 `S::new()` 的调用所示：

```
/// 返回document中不在wordlist中的词的集合
fn unknown_words<S: StringSet>(document: &Vec<String>, wordlist: &S) -> S {
    let mut unknowns = S::new();
    for word in document {
        if !wordlist.contains(word) {
            unknowns.add(word);
        }
    }
    unknowns
}
```

与 Java 和 C# 的接口类似，特型目标不支持静态方法。如果想要使用特型目标 `&StringSet`，就必须修改特型，给每个静态方法都添加 `where Self: Sized` 绑定：

```
trait StringSet {
    fn new() -> Self
        where Self: Sized;

    fn from_slice(strings: &[&str]) -> Self
        where Self: Sized;

    fn contains(&self, string: &str) -> bool;

    fn add(&mut self, string: &str);
}
```

这个绑定告诉 Rust：特型目标将免于支持这个方法。然后，就可以使用 `&StringSet` 特型目标了。特型目标仍然不支持两个静态方法，但你可以创建它们并用于调用 `.contains()` 和 `.add()`。同样的做法也适用于任何其他不兼容特型目标（不能与特型目标共存）的方法。（这里就不解释为什么这样能行的技术细节了，不过第 13 章会介绍 `Sized` 特型。）

11.3　完全限定方法调用

方法其实就是一种特殊的函数。以下两种方法调用是等价的：

```
"hello".to_string()

str::to_string("hello")
```

第二种形式看起来就是一个静态方法调用。即使 `to_string` 方法接收 `self` 参数，这样调用也没有问题。只要把 `self` 作为函数的第一个参数传给它即可。

由于 `to_string` 是标准 `ToString` 特型的方法，因此还有两种形式可以使用：

```
ToString::to_string("hello")

<str as ToString>::to_string("hello")
```

以上 4 种方法调用都完成同样的操作。一般情况下，还是 `value.method()` 这种形式用的比较多。其他形式称为**限定**方法调用，因为它们需要指定方法关联的类型或特型。最后一种带尖括号的形式，则同时指定了两者，因此称为**完全限定**方法调用。

在 `"hello".to_string()` 这种形式中，使用的是 `.` 操作符，并没有确切说明要调用的是哪个 `to_string` 方法。Rust 会通过一个方法查询算法去找，根据类型、强制解引用（deref coercion）等来判断。完全限定调用要确切指定使用的方法，这在一些边界情况下很有用。

- 两个方法同名。经典的例子是从两个不同特型得到两个 `.draw()` 方法的 `Outlaw`，其中一个可以把它绘制到屏幕上，另一个用于挑战法律。

  ```
  outlaw.draw();  // 错误：绘制到屏幕还是开枪？

  Visible::draw(&outlaw); // 可以：绘制到屏幕
  HasPistol::draw(&outlaw); // 可以：抓住他
  ```

  正常情况下应该有机会重命名其中一个方法，但有时候确实不行。

- 无法推断 `self` 参数的类型。

  ```
  let zero = 0; // 类型未指定，可能是i8、u8……

  zero.abs(); // 错误：没有找到abs方法
  i64::abs(zero); // 可以
  ```

- 将函数本身作为值。

  ```
  let words: Vec<String> =
      line.split_whitespace() // 产生&str值的迭代器
          .map(<str as ToString>::to_string) // 可以
          .collect();
  ```

  这里的完全限定形式 `<str as ToString>::to_string` 就是一种给传入 `.map` 的函数命名的方式。

- 在宏里调用特型方法。第 20 章将解释。

完全限定语法也适用于静态方法。上一节用 S::new() 在一个泛型函数中创建了一个新集合。实际上也可以写成 StringSet::new() 或 <S as StringSet>::new()。

11.4 定义类型关系的特型

到目前为止，本书介绍的每一种特型似乎都是独立的：特型是类型可以实现的一组方法。实际上，在需要多个类型相互协作的情况下，特型同样能派上用场，因为特型可以用来描述类型之间的关系。

- std::iter::Iterator 特型通过自己产生值的类型将不同的迭代器类型关联起来。
- std::ops::Mul 特型关联可以参与乘法运算的类型。在表达式 a * b 中，a 和 b 这两个值的类型可以相同，也可以不同。
- rand 包中既包含一个跟随机数生成器有关的特型（rand::Rng），还包含一个与能够被随机生成的类型有关的特型（rand::Rand）。这两个特型实际上就定义了实现它们的类型之间的协作关系。

平时做开发并不需要经常创建类似这样的特型。问题是你在使用标准库和第三方包的时候，总会看到它们的身影。因此本节就来分析一下上面每个例子背后的实现，然后在分析过程中顺便也把相关的 Rust 语言特性介绍一下。关键是要培养自己阅读特型和方法签名的能力，做到一看就知道它们对相关类型有什么要求。

11.4.1 关联类型（或迭代器工作原理）

先从迭代器开始吧。现在，几乎所有面向对象语言都会内置支持某种形式的迭代器。所谓迭代器，就是一个能够通过它遍历一系列值的对象。

Rust 有一个标准的 Iterator 特型，定义如下：

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

这个特型的第一个特性 type Item; 是一个**关联类型**（associated type）。所有实现 Iterator 的类型都必须指定自己产生的项（item）的类型。

第二个特性是 next() 方法，在返回值中使用了这个关联类型。next() 返回一个 Option<Self::Item>，即要么是 Some(item)，其中 item 是序列中的下一个值，要么是 None，表明已经没有更多值了。这个类型写作 Self::Item，而不是单纯的 Item，是因为 Item 是每个迭代器类型的特性，而不是某个独立类型的特性。一如既往，self 和 Self 类型在代码中依旧会出现在它们的字段、方法等被用到的地方。

下面来看一个实现 Iterator 的类型：

```
//（截取自std::env标准库模块的代码）
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

std::env::Args 是标准库函数 std::env::args() 返回的迭代器类型，第 2 章曾使用它访问过命令行参数。这个迭代器产生 String 值，因此 impl 块声明了 type Item = String;。

泛型代码可以使用关联类型：

```
/// 遍历一个迭代器，将它们的值存储到一个新向量中
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

在这个函数体内，Rust 会为我们推断 value 的类型，这当然很好。不过，我们必须写出 collect_into_vector 的返回值类型，而关联类型 Item 是描述返回值类型的唯一方式。（Vec<I> 肯定不对，这样就是声明返回迭代器向量了！）

前面例子中的代码并不需要自己编写，看完第 15 章之后你会知道迭代器已经有了一个实现同样操作的标准方法：iter.collect()。既然说到这儿了，那索性就多看一个例子，然后再往下讲。

```
/// 打印出一个迭代器产生的所有值
fn dump<I>(iter: I)
    where I: Iterator
{
    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value); // 错误
    }
}
```

其实就差一点。这里只有一个问题：value 可能不是可打印类型。

```
error[E0277]: the trait bound `<I as std::iter::Iterator>::Item:
              std::fmt::Debug` is not satisfied
  --> traits_dump.rs:10:37
   |
10 | println!("{}: {:?}", index, value);   // 错误
   |                             ^^^^^ the trait `std::fmt::Debug`
   |                                   is not implemented for
   |                                   `<I as std::iter::Iterator>::Item`
   |
   = help: consider adding a
           `where <I as std::iter::Iterator>::Item: std::fmt::Debug` bound
   = note: required by `std::fmt::Debug::fmt`
```

由于 Rust 使用了 <I as std::iter::Iterator>::Item 这种语法，导致上面的报错信息不太好理解。其实这是表达 I::Item 的最长、最明确的形式。虽然是有效的 Rust 语法，但实际上很少需要这样写。

报错消息的要点在于，如果想让泛型函数通过编译，那必须保证 I::Item 实现 Debug 特型，也就是以 {:?} 来格式化值的特型。为此，可以在 I::Item 上增加一个绑定：

```
use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}
```

或者，也可以声明“I 必须是一个遍历 String 值的迭代器”：

```
fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}
```

这里的 Iterator<Item=String> 本身就是一个特型。如果认为 Iterator 是所有迭代器类型的一个集合，那么 Iterator<Item=String> 就是 Iterator 的一个子集，即产生 String 的迭代器类型的集合。在任何可以使用特型名字的地方都可以使用这种语法，包括特型目标类型：

```
fn dump(iter: &mut Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{}: {:?}", index, s);
    }
}
```

像 Iterator 这种带有关联类型的特型是可以与特型方法共存的，但前提是必须把所有关联类型都明确写出来，就像示例中所展示的那样。否则，这个示例中的 s 就可以是任意类型，于是 Rust 就没办法对代码做类型检查了。

前面已经展示了很多迭代器的例子。事实上，也很难不举这些例子，毕竟关联类型最重要的用武之地就是迭代器。不过，关联类型本身在特型需要覆盖除方法之外的定义时也是很有用的。

- 一个线程池的库中的 Task 特型（表示工作单元），可以包含一个关联的 Output 类型。
- 一个 Pattern 特型（表示搜索字符串的一种方式），可以包含一个关联的 Match 类型，表示模式与字符串匹配之后收集到的所有信息。

```
trait Pattern {
    type Match;

    fn search(&self, string: &str) -> Option<Self::Match>;
}
```

```
/// 可以在字符串中搜索特定的字符
impl Pattern for char {
    /// Match（匹配）表示的就是发现字符的位置
    type Match = usize;

    fn search(&self, string: &str) -> Option<usize> {
        ...
    }
}
```

如果熟悉正则表达式，很容易看出来 `impl Pattern for RegExp` 应该可以把 `Match` 类型声明为一个更复杂的类型，比如一个包含匹配起止位置、捕获组位置等信息的结构体。

- 一个操作关系数据库的库可以有一个 `DatabaseConnection` 特型，而这个特型可以有表示事务、指针、初始化语句等的关联类型。

关联类型非常适合每个实现都有**一个**特定的相关类型的情况。比如，每个 `Task` 类型产生一个特定的 `Output` 类型，每个 `Pattern` 类型查找一个特定的 `Match` 类型。然而，马上要看到的类型间的某些关系并不是这样的。

11.4.2　泛型特型（或操作符重载的原理）

Rust 中的乘法运算使用这个特型：

```
/// std::ops::Mul，支持*操作符的类型实现的特型
pub trait Mul<RHS> {
    /// 应用*操作符之后返回的结果类型
    type Output;

    /// *操作符对应的方法
    fn mul(self, rhs: RHS) -> Self::Output;
}
```

`Mul` 是一个泛型特型。类型参数 `RHS` 是 Right Hand Side（**右手边**）的简写形式。

类型参数在这里的含义与它在结构体或者函数中的含义相同。具体来说，`Mul` 是一个泛型特型，其实例 `Mul<f64>`、`Mul<String>`、`Mul<Size>` 等，都是不同的特型，就像 `min::<i32>` 和 `min::<String>` 是不同的函数，`Vec<i32>` 和 `Vec<String>` 是不同的类型一样。

一个类型，比如 `WindowSize`，可以既实现 `Mul<f64>` 又实现 `Mul<i32>`，甚至实现更多。然后，就可以将一个 `WindowSize` 跟很多其他类型相乘了。每个实现都会有自己关联的 `Output` 类型。

上面代码中展示的特型少了一个小细节。真正的 `Mul` 特型是这样的：

```
pub trait Mul<RHS=Self> {
    ...
}
```

语法 `RHS=Self` 的意思是 `RHS` 默认为 `Self`。如果我写 `impl Mul for Complex`，不指定 `Mul` 的类型参数，那就相当于写的是 `impl Mul<Complex> for Complex`。在绑定中，如果我写 `where T: Mul`，则相当于写的是 `where T: Mul<T>`。

在 Rust 中，表达式 `lhs * rhs` 是 `Mul::mul(lhs, rhs)` 的简写形式。因此，在 Rust 中重载 `*` 操作符和实现 `Mul` 特型一样简单。具体的例子下一章会详细介绍。

11.4.3 伴型特型（或rand::random()工作原理）

还有一种使用特型表达类型之间关系的方式。这种方式可能是最简单的，因为理解它不必学习任何新的语言特性。这里所说的**伴型特型**（buddy trait），其实就是设计用来协同工作的特型。

颇受欢迎的生成随机数的 `rand` 包中有一个不错的例子。`rand` 的主要功能是 `random()` 函数，它返回一个随机值：

```
use rand::random;
let x = random();
```

如果 Rust 无法推断随机值的类型（经常会有这种情况），那你必须指定：

```
let x = random::<f64>();  // 数值，0.0 ≤ x < 1.0
let b = random::<bool>(); // true或false
```

对于很多程序来说，这种泛型函数就足够了。不过 `rand` 包中也提供了几种不同但可以互操作的随机数生成器。这个包中的所有随机数生成器都实现了一个公共特型：

```
/// 一个随机数生成器
pub trait Rng {
    fn next_u32(&mut self) -> u32;
    ...
}
```

`Rng` 就是一个可以根据需求输出整数的值。`rand` 包提供了几种对它不同的实现，包括 `XorShiftRng`（一个快速伪随机数生成器）和 `OsRng`（要慢得多，但真的无法预测，用于加密）。

这里用到的伴型特型叫 `Rand`：

```
/// 可以使用Rng随机生成的类型
pub trait Rand: Sized {
    fn rand<R: Rng>(rng: &mut R) -> Self;
}
```

`f64` 和 `bool` 等类型实现了这个特型。给它们的 `::rand()` 方法传入任意随机数生成器，都会生成一个随机值：

```
let x = f64::rand(rng);
let b = bool::rand(rng);
```

实际上，`random()` 就是简单包装了一下对这个 `rand` 方法的调用，并传给它一个全局分配的 `Rng`。以下是实现它的一种方式：

```
pub fn random<T: Rand>() -> T {
    T::rand(&mut global_rng())
}
```

看到使用其他特型作为绑定的特型（也就是 Rand::rand() 使用 Rng 的方式），就会知道这两个特型是相辅相成的。换句话说，任意 Rng 可以生成所有 Rand 类型的值。因为这个方法涉及泛型，所以 Rust 会为程序中用到的每种 Rng 与 Rand 的组合生成优化的机器码。

这两种特型也各自有不同的关注点。如图 11-3 所示，无论是为 Monster 类型实现 Rand，还是实现一个特别快但没那么随机的 Rng，都不必操心它们怎样相互配合，除了实现什么也不用做。

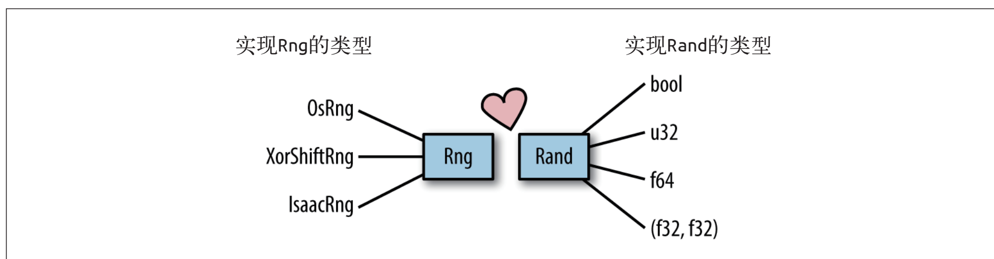


图 11-3：伴型特型示意图。左侧的 Rng 类型是 rand 包提供的真正的随机数生成器

标准库中计算散列码的实现是伴型特型的另一个例子。实现 Hash 的类型是可以散列化的，因此可以用作散列表的键。实现 Hasher 的类型是散列化算法。这两个特型以跟 Rand 和 Rng 相同的方式连接在一起：Hash 有一个泛型方法 Hash::hash()，接收任何类型的 Hasher 作为参数。

还有一个例子是 serde 库的 Serialize 特型，11.2.2 节曾经介绍过。但它有一个伴型特型前面没提到，就是 Serializer 特型，它表示输出格式。serde 支持可插拔的序列化格式。不同的 Serializer 实现了 JSON、YAML、一个叫 CBOR 的二进制格式，等等。由于这两个特型之间的紧密协作关系，每种格式都自动支持每种可序列化的类型。

最后这 3 节讨论了用特型描述类型之间关系的 3 种方式。所有这些都可以看成为了避免虚拟方法消耗和向下转型，因为它们可以让 Rust 在编译时知道更具体的类型。

11.5　逆向工程绑定

如果没有一个特型可以满足你所有的需求，那写泛型代码是非常困难的。假设我们已经写了下面这个非泛型函数以实现某些计算：

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

现在想重用同一套代码去计算浮点值。那么可以尝试这样改：

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total: N = 0;
    for i in 0 .. v1.len() {
```

```
        total = total + v1[i] * v2[i];
    }
    total
}
```

很不幸，Rust 会抱怨代码中使用了 + 和 *，还有 0 的类型。可以使用 `Add` 和 `Mul` 特型，要求 `N` 是一个支持 + 和 * 的类型。这里用到的 0 也要改，因为 Rust 中的 0 始终是整数，对应的浮点值是 0.0。这里恰好有一个标准的特型 `Default` 可以让有默认值的类型使用。对于数值类型，默认值始终为 0。

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

已经很接近了，但还是差那么一点：

```
error[E0308]: mismatched types
  --> traits_generic_dot_2.rs:11:25
   |
11 | total = total + v1[i] * v2[i];
   |                 ^^^^^^^^^^^^^ expected type parameter, found associated type
   |
   = note: expected type `N`
              found type `<N as std::ops::Mul>::Output`
```

我们的新代码假设乘以两个类型 `N` 的值会得到另一个类型 `N` 的值，但这可是不一定的。此时可以重载乘法操作符返回想要的任何类型。我们需要一种方式告诉 Rust 这个泛型函数只适用于常规的乘法运算，即 `N * N` 就返回 `N`。为此，要把 `Mul` 替换成 `Mul<Output=N>`。`Add` 也采用同样的改法。

```
fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -> N
{
    ...
}
```

这样一改，加入了很多绑定，代码的可读性立马下降了很多。那把绑定转移到 `where` 子句中：

```
fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}
```

不错。但 Rust 依旧抱怨这行代码有问题：

```
error[E0508]: cannot move out of type `[N]`, a non-copy array
 --> traits_generic_dot_3.rs:7:25
  |
7 |         total = total + v1[i] * v2[i];
  |                         ^^^^^ cannot move out of here
```

这个报错可真难理解，就算我们都已经熟悉了相关概念也看不懂。没错，把 v1[i] 的值转移出切片是不合法。但数值不是可复制的吗？又是哪里出问题了？

答案是 Rust **不知道** v1[i] 是一个数值。事实上，它确实不是。基于目前给出的绑定信息，N 可能是任何类型。如果也想让 N 是一个可复制类型，必须再加一个绑定：

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

加完以后，代码就可以编译运行了。最终代码如下所示：

```
use std::ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 88.0);
}
```

刚刚这个情景在编写 Rust 程序时偶有发生：先是跟编译器紧张地争论一番，最终代码变成了很好看的样子，就好像是没费吹灰之力就写出来的，而且运行得也很“丝滑”。

我们在这里所做的是对 N 的绑定进行了逆向工程，让编译器当指导，反复检查自己的工作。之所以这个过程有点痛苦，主要原因是标准库中并没有那么一个 Number 特型，包含我们想要使用的所有操作符和方法。真是无巧不成书，有一个流行的开源 Rust 包叫 num，它就定义了这么一个特型！如果早一点知道，就可以在 Cargo.toml 中加上 num，然后这样写：

```
use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

就像在面向对象编程中一样，正确的接口能让一切变美好。而在泛型编程中，正确的特型能让一切变美好。

还是那个问题，为什么要这么麻烦呢？为什么 Rust 的设计者不把泛型设计得更像 C++ 模板，让所有约束都隐含在代码中，模仿“鸭子类型”？

Rust 的这种设计的一个优点是可以让泛型代码具有向前兼容的能力。你可以修改一个公有泛型函数或方法的实现，只要不改变签名，就不会给它的任何用户造成麻烦。

绑定的另一个优点在于，你能通过编译器报错知道要解决的麻烦在哪里。C++ 编译器涉及模板的错误消息要比 Rust 的长很多，并且会指出很多不同的行号来，因为编译器没办法知道对于发生的问题该责怪谁。是模板呢，还是它的调用者？而调用者可能也是一个模板；当然，**那个**模板也可能是个调用者……

或许，明确写出绑定最重要的好处是它们在代码和文档里都存在。这样只要你一看到 Rust 泛型函数的签名，就能确切知晓它接收什么样的参数。使用模板则做不到这一点。与我们在这里经历的挫折相比，Boost 之类的 C++ 库要完整写出所有类型的文档，那才真叫受罪呢。可没有编译器帮 Boost 的开发者检查它们的工作。

11.6　小结

特型是 Rust 中一种主要的组织性手段，而且其存在有充分的必要性。设计程序或库最重要的就是设计出好的接口。

本章，我们经历了一波语法、规则和解释的轮番冲击。如果你仍然屹立不倒，那么接下来本书就可以讨论在 Rust 代码中使用特型和泛型的各种方式了。悄悄地告诉你，本章只不过是一场头脑风暴的前奏而已。接下来的两章将介绍标准库提供的常用特型。之后的各章将讨论到闭包、迭代器、输入 / 输出和并发，而特型和泛型在这些主题中都扮演了重要角色。

第12章

操作符重载

> 关于数学的确切范围和定义，数学家和哲学家有不同的看法……但他们的看法都有严重的问题，没有得到广泛接受，看起来也没有和解的可能。
>
> ——维基百科，关于“数学”的定义

第 2 章在绘制曼德布洛特集合的例子中使用了 num 包的 Complex 类型来表示复平面上的点：

```
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// 复数的实数部分
    re: T,

    /// 复数的虚数部分
    im: T
}
```

使用 Rust 的 + 和 * 操作符，可以像计算内置数值类型一样对 Complex 值进行加和乘：

```
z = z * z + c;
```

你也可以让自己定义的类型支持算术和其他操作，只要实现几个内置特型即可。这就称为**操作符重载**，其效果跟 C++、C#、Python 和 Ruby 中的操作符重载很像。

操作符重载的特型可以分为几类，具体取决于它们支持 Rust 语言的哪个部分，如表 12-1 所示。本章接下来几节会依次介绍这几类特型。

表12-1：支持操作符重载的特型

类　别	特　型	操　作　符
一元操作符	`std::ops::Neg`	`-x`
	`std::ops::Not`	`!x`
算术操作符	`std::ops::Add`	`x + y`
	`std::ops::Sub`	`x - y`
	`std::ops::Mul`	`x * y`
	`std::ops::Div`	`x / y`
	`std::ops::Rem`	`x % y`
位操作符	`std::ops::BitAnd`	`x & y`
	`std::ops::BitOr`	`x | y`
	`std::ops::BitXor`	`x ^ y`
	`std::ops::Shl`	`x << y`
	`std::ops::Shr`	`x >> y`
复合赋值算术操作符	`std::ops::AddAssign`	`x += y`
	`std::ops::SubAssign`	`x -= y`
	`std::ops::MulAssign`	`x *= y`
	`std::ops::DivAssign`	`x /= y`
	`std::ops::RemAssign`	`x %= y`
复合赋值位操作符	`std::ops::BitAndAssign`	`x &= y`
	`std::ops::BitOrAssign`	`x |= y`
	`std::ops::BitXorAssign`	`x ^= y`
	`std::ops::ShlAssign`	`x <<= y`
	`std::ops::ShrAssign`	`x >>= y`
比较	`std::cmp::PartialEq`	`x == y`、`x != y`
	`std::cmp::PartialOrd`	`x < y`、`x <= y`、`x > y`、`x >= y`
索引	`std::ops::Index`	`x[y]`、`&x[y]`
	`std::ops::IndexMut`	`x[y] = z`、`&mut x[y]`

12.1　算术与位操作符

在 Rust 中，表达式 `a + b` 实际上是 `a.add(b)` 的简写，即这是对标准库中 `std::ops::Add` 特型的 `add` 方法的调用。Rust 标准的数值类型全部实现了 `std::ops::Add`。为了让表达式 `a + b` 能够用于 `Complex` 值，`num` 包也为 `Complex` 实现了这个特型。类似的特型也对应着其他操作符。比如，`a * b` 是对 `a.mul(b)` 的简写，也就是对 `std::ops::Mul` 特型方法的调用；同理，`std::ops::Neg` 对应于取反操作符 `-`，等等。

如果非要写成 `z.add(c)` 的形式，那需要把 `Add` 特型引入作用域，以便它的方法可见。这样一来，就可以把所有算术操作当成函数调用：[1]

```
use std::ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

注 1：Lisp 程序员很高兴。表达式 `::add` 是 `i32` 的 `+` 操作符，其作为函数值被捕获。

以下是 std::ops::Add 的定义：

```
trait Add<RHS=Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

换句话说，特型 Add<T> 代表给自己的类型加上一个 T 值的能力。比如，如果想让自己的类型可以加 i32 和 u32 值，那你的类型必须实现 Add<i32> 和 Add<u32>。特型的类型参数 RHS 默认值为 Self，实现两个相同类型值的加法比较简单，直接实现 Add 就行。关联类型 Output 描述相加的结果。

比如，为了实现两个 Complex<i32> 值相加，Complex<i32> 必须实现 Add<Complex<i32>>。因为是在同类型自身基础上相加，所以只需写 Add：

```
use std::ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

当然，没必要分别为 Complex<i32>、Complex<i64>、Complex<f32>、Complex<f64> 单独实现 Add。因为它们除了类型不一样，其他定义都相同，所以应该可以只写一个普适的泛型实现，只要复数组件本身支持相加就行：

```
use std::ops::Add;

impl<T> Add for Complex<T>
    where T: Add<Output=T>
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

通过写入 where T: Add<Output=T> 将 T 限制为可以与自身相加的类型，而相加产生的是另一个 T 值。这个限制虽然合理，但还是可以把条件放宽一点，毕竟 Add 特型不要求 + 的两个操作数类型相同，也没有限制结果类型。而最宽松的泛型实现莫过于让左、右操作数的类型互不影响，同时能够产生一个包含加法操作所得到组件类型的 Complex 值：

```
use std::ops::Add;

impl<L, R, O> Add<Complex<R>> for Complex<L>
    where L: Add<R, Output=O>
{
    type Output = Complex<O>;
    fn add(self, rhs: Complex<R>) -> Self::Output {
        Complex { re: self.re + rhs.re, im: self.im + rhs.im }
    }
}
```

不过，实践中 Rust 倾向于避免支持混合类型操作。因为类型参数 L 必须实现 Add<R, Output=O>，所以 L、R 和 O 通常都为同一类型。原因很简单，不太可能有那么多类型让 L 全都实现了。最终，这个最宽松的泛型函数可能也不会比前面那个更简单的版本更有用。

Rust 为算术和位操作符而内置的特型分为 3 组：一元操作符、二元操作符和复合赋值操作符。在每一组中，特型及其方法都有相同的形式，因此接下来每组会举一个例子。

12.1.1　一元操作符

除了将在 13.5 节单独介绍的引用操作符 *，Rust 还有两种一元操作符可以让我们自定义，如表 12-2 所示。

表12-2：一元操作符的内置特型

特　型　名	表　达　式	等价表达式
std::ops::Neg	-x	x.neg()
std::ops::Not	!x	x.not()

Rust 的所有数值类型都实现了 std::ops::Neg，以支持一元取反操作符 -。整数类型和 bool 还实现了 std::ops::Not，以支持一元非操作符 !。同时，它们也实现了对这些类型的引用。

注意，! 操作符对 bool 值取非，对整数执行按位非（即按位反转），相当于 C 和 C++ 中的 ! 和 ~ 这两个操作符。

这两个特型的定义很简单：

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

对复数值取反就是简单地对其每个组件取反。以下是对 Complex 值取反的一个泛型实现：

```
use std::ops::Neg;

impl<T, O> Neg for Complex<T>
    where T: Neg<Output=O>
{
    type Output = Complex<O>;
    fn neg(self) -> Complex<O> {
        Complex { re: -self.re, im: -self.im }
    }
}
```

12.1.2　二元操作符

Rust 二元算术和位操作符及它们对应的内置特型如表 12-3 所示。

表12-3：二元操作符的内置特型

类　别	特　型　名	表　达　式	等价表达式
算术操作符	`std::ops::Add`	`x + y`	`x.add(y)`
	`std::ops::Sub`	`x - y`	`x.sub(y)`
	`std::ops::Mul`	`x * y`	`x.mul(y)`
	`std::ops::Div`	`x / y`	`x.div(y)`
	`std::ops::Rem`	`x % y`	`x.rem(y)`
位操作符	`std::ops::BitAnd`	`x & y`	`x.bitand(y)`
	`std::ops::BitOr`	`x | y`	`x.bitor(y)`
	`std::ops::BitXor`	`x ^ y`	`x.bitxor(y)`
	`std::ops::Shl`	`x << y`	`x.shl(y)`
	`std::ops::Shr`	`x >> y`	`x.shr(y)`

Rust 的所有数值类型都实现了算术操作符。Rust 的整数类型和 `bool` 实现了位操作符。当然，它们也实现了接受对这些类型的引用作为一个或两个操作数的逻辑。

所有上述特型都有统一的形式。比如，针对 ^ 操作符的 `std::ops::BitXor` 的定义如下：

```
trait BitXor<RHS=Self> {
    type Output;
    fn bitxor(self, rhs: RHS) -> Self::Output;
}
```

本章开头也展示了同类别的另一个特型 `std::ops::Add`，以及几个简单的实现。

特型 `Shl` 和 `Shr` 跟这个模式稍微有点不同：它们没有默认指定 `RHS` 类型参数的值为 `Self`，因此重新实现时必须明确给出右操作数的类型。`<<` 和 `>>` 操作符的右操作数代表移多少位，跟要移位的值的类型没太大关系。

使用 + 操作符可以将一个 `String` 和一个 `&str` 切片或另一个 `String` 拼接起来。但是，Rust 不允许 + 的左操作数是 `&str`，目的是阻止通过重复小的左操作数来构建长字符串。（这种操作存在性能隐患，所需时间与最终字符串长度的平方正相关。）通常，要一段一段地拼接字符串，最好使用 `write!`，17.3.3 节将介绍怎么做。

12.1.3　复合赋值操作符

复合赋值表达式类似于 `x += y` 或 `x &= y`，即接受两个操作数，先对它们执行加法或按位与操作，再把结果保存到左操作数中。在 Rust 中，复合赋值表达式的值始终是 `()`，而不是左操作数最终保存的值。

很多语言有类似的操作符，且通常都将它们定义为表达式 `x = x + y` 或 `x = x & y` 的简写形式。然而，Rust 没有遵循这个惯例。在 Rust 中，`x += y` 是方法调用 `x.add_assign(y)` 的简写形式，其中 `add_assign` 是 `std::ops::AddAssign` 特型的唯一方法。

```
trait AddAssign<RHS=Self> {
    fn add_assign(&mut self, RHS);
}
```

表 12-4 展示了 Rust 的所有复合赋值操作符，以及实现它们的内置特型。

表12-4：复合赋值操作符的内置特型

类　别	特　型　名	表　达　式	等价表达式
算术操作符	`std::ops::AddAssign`	`x += y`	`x.add_assign(y)`
	`std::ops::SubAssign`	`x -= y`	`x.sub_assign(y)`
	`std::ops::MulAssign`	`x *= y`	`x.mul_assign(y)`
	`std::ops::DivAssign`	`x /= y`	`x.div_assign(y)`
	`std::ops::RemAssign`	`x %= y`	`x.rem_assign(y)`
位操作符	`std::ops::BitAndAssign`	`x &= y`	`x.bitand_assign(y)`
	`std::ops::BitOrAssign`	`x |= y`	`x.bitor_assign(y)`
	`std::ops::BitXorAssign`	`x ^= y`	`x.bitxor_assign(y)`
	`std::ops::ShlAssign`	`x <<= y`	`x.shl_assign(y)`
	`std::ops::ShrAssign`	`x >>= y`	`x.shr_assign(y)`

Rust 的所有数值类型都实现了算术复合赋值操作符。Rust 的整数类型和 `bool` 还实现了位复合赋值操作符。

对 `Complex` 类型进行 `AddAssign` 的泛型实现也很简单：

```
use std::ops::AddAssign;

impl<T> AddAssign for Complex<T>
    where T: AddAssign<T>
{
    fn add_assign(&mut self, rhs: Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}
```

复合操作符的内置特型与对应的二元操作符的内置特型完全是相互独立的。实现 `std::ops::Add` 不会自动实现 `std::ops::AddAssign`。如果你想让 Rust 允许你的自定义类型作为 `+=` 操作符的左操作数，就必须自己实现 `AddAssign`。

与二元 `Shl` 和 `Shr` 特型类似，`ShlAssign` 和 `ShrAssign` 特型也与其他复合赋值特型不太一样：它们没有将 `RHS` 类型参数默认为 `Self`，所以实现时必须明确给出右操作数的类型。

12.2　相等测试

Rust 的相等操作符 `==` 和 `!=` 是对调用 `std::cmp::PartialEq` 特型的方法 `eq` 和 `ne` 的简写：

```
assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));
```

以下是 `std::cmp::PartialEq` 的定义：

```
trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { !self.eq(other) }
}
```

因为 ne 方法有一个默认的定义，所以只需定义 eq 就可以实现 PartialEq 特型。下面是 Complex 的完整实现：

```
impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) -> bool {
        self.re == other.re && self.im == other.im
    }
}
```

换句话说，对于任何自身可以比较相等的组件类型 T，这里为 Complex<T> 实现了比较。假设我们也为 Complex 实现了 std::ops::Mul，那么就可以这样写：

```
let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im: 29 });
```

PartialEq 的实现基本上就是这里展示的形式，即同时比较左操作数和右操作数的每个对应字段。这个实现写起来有点麻烦，比较相等则是 Rust 要支持的常见操作，所以如果你要求，Rust 会为你自动生成 PartialEq 的实现。怎么要求呢？只要像下面这样把 PartialEq 添加到 derive 属性的类型定义中即可：

```
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}
```

Rust 自动生成的实现与我们手工编写的代码基本相同，都是依次比较相应类型的每个字段或元素。Rust 也可以为 enum 类型派生 PartialEq 实现。自然地，这个类型拥有（或者对 enum 而言是可能拥有）的每个值自身也必须实现 PartialEq。

与算术或位操作相关的特型会取得操作数的值，但 PartialEq 会取得操作数的引用。这意味着对 String、Vec 或 HashMap 等非 Copy 值的比较不会导致它们的值转移，而这可能会带来麻烦：

```
let s = "d\x6fv\x65t\x61i\x6c".to_string();
let t = "\x64o\x76e\x74a\x69l".to_string();
assert!(s == t);  // s和t在此只是被借用……

// ……因此在这里它们仍然拥有自己的值
assert_eq!(format!("{} {}", s, t), "dovetail dovetail");
```

这里就不得不提到这个特型对 Rhs 类型参数的绑定了，这种类型参数是之前没见过的：

```
Rhs: ?Sized
```

这样写相当于放宽了 Rust 对类型参数必须有大小的限制，因而才能写出 PartialEq<str> 或 PartialEq<[T]> 这样的特型。方法 eq 和 ne 接收 &Rhs 类型的参数，比较 &str 或 &[T] 是完全合理的。由于 str 实现了 PartialEq<str>，因此下列断言都是相等的：

```
assert!("ungula" != "ungulate");
assert!("ungula".ne("ungulate"));
```

在这里，`Self` 和 `Rhs` 可能是非固定大小的 `str`，从而让 `ne` 的 `self` 和 `rhs` 参数都是 `&str` 值。关于固定大小的类型、非固定大小的类型，以及 `Sized` 特型，13.2 节会详细介绍。

为什么这个特型叫 `PartialEq` 呢？传统数学对**相等关系**（这里相等是一种情形）的定义包含 3 方面要求。对于任意值 `x` 和 `y`，需满足以下条件。

- 如果 `x == y` 是 `true`，那么 `y == x` 也必须是 `true`。换句话说，交换比较相等操作的两个操作数不影响比较结果。
- 如果 `x == y` 且 `y == z`，那么 `x == z` 也一定是 `true`。对于任何连续相等的值，其中任意一个值都与其他值直接相等。相等具有传递性。
- `x == x` 永远是 `true`。

第三个要求似乎很明显，都不用说出来了，但这条规则正是很多问题的根源。Rust 的 `f32` 和 `f64` 是 IEEE 标准浮点值。根据该标准，`0.0/0.0` 或其他没有适当值的表达式必须产生特殊的“Not a Number”（非数值）值，通常称为 `NaN` 值。这个标准进一步要求 `NaN` 值必须不跟其他任何值相等，包括它自己。比如，该标准要求所有下列断言为 `true`：

```
assert!(f64::is_nan(0.0/0.0));
assert_eq!(0.0/0.0 == 0.0/0.0, false);
assert_eq!(0.0/0.0 != 0.0/0.0, true);
```

不仅如此，任何与 `NaN` 的顺序比较都必须返回 `false`：

```
assert_eq!(0.0/0.0 < 0.0/0.0, false);
assert_eq!(0.0/0.0 > 0.0/0.0, false);
assert_eq!(0.0/0.0 <= 0.0/0.0, false);
assert_eq!(0.0/0.0 >= 0.0/0.0, false);
```

Rust 的 `==` 操作符符合 IEEE 对相等关系的前两个要求，而在涉及 IEEE 浮点值时明显不符合第三个条件。这就叫作**部分相等关系**（partial equivalence relation），所以 Rust 使用 `PartialEq` 这个名字为 `==` 操作符定义了内置的特型。如果你知道自己写的泛型代码中的类型参数只能是 `PartialEq`，那么可以假定前两个要求有效，但不能假定一个值始终与它自身相等。

这多少有点违反直觉，一不小心还可能导致错误。如果你的泛型代码需要保证完全相等的关系，那可以使用 `std::cmp::Eq` 特型作为绑定，它表示完全相等关系。换句话说，如果一个类型实现了 `Eq`，那么这个类型的每个值 `x`，一定有 `x == x` 为 `true`。在实践中，几乎所有实现 `PartialEq` 的类型都应该实现 `Eq`。`f32` 和 `f64` 是标准库中仅有的两个为 `PartialEq` 而非 `Eq` 的类型。

标准库将 `Eq` 定义为 `PartialEq` 的扩展，而且没有定义新方法：

```
trait Eq: PartialEq<Self> { }
```

如果你的类型是 `PartialEq`，那么也应该希望它是 `Eq`。为此，必须再明确实现 `Eq`，尽管此实现并不真的需要定义任何新函数或类型。因此，为 `Complex` 实现 `Eq` 非常简单：

```
impl<T: Eq> Eq for Complex<T> { }
```

甚至，在 Complex 类型定义的 derive 属性中包含 Eq 就足以实现它了：

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

对泛型类型的派生实现可能会因类型参数不同而不同。有了 derive 属性，Complex<i32> 会实现 Eq，因为 i32 可以完全相等，而 Complex<f32> 只会实现 PartialEq，因为 f32 无法实现 Eq。

如果你自己实现 std::cmp::PartialEq，Rust 则无法检查你对 eq 和 ne 方法的定义与对部分或全部相等的要求完全一致。如何实现它们是你的事，Rust 相信你会以符合特型用户预期的方式实现相等测试。

虽然 PartialEq 提供了 ne 的默认定义，但如果愿意，你也可以自己重新实现。不过，重新实现必须确保 eq 和 ne 互相可逆。PartialEq 特型的用户会认为这是自然而然的。

12.3 顺序比较

Rust 通过一个特型 std::cmp::PartialOrd 规定了顺序比较操作符 <、>、<= 和 >= 的行为：

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs> where Rhs: ?Sized {
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

没错，PartialOrd<Rhs> 扩展了 PartialEq<Rhs>。这意味着只能对可以进行相等比较的类型做顺序比较。

PartialOrd 特型唯一需要实现的方法是 partial_cmp。如果 partial_cmp 返回 Some(o)，则 o 表示 self 与 other 的关系：

```
enum Ordering {
    Less,       // self < other
    Equal,      // self == other
    Greater,    // self > other
}
```

但如果 partial_cmp 返回 None，那就意味着 self 和 other 相互之间无法区分顺序：既不是谁大于谁，也不是谁等于谁。在 Rust 所有的原始类型中，只有浮点值之间的比较可能返回 None。特别地，比较 NaN（非数值）与任何其他值都返回 None。12.2 节对 NaN 值进行过介绍。

与其他二元操作符类似，要比较两个类型 Left 和 Right 的值，Left 必须实现 PartialOrd<Right>。如表 12-5 所示，表达式 x < y 和 x >= y 都是对 PartialOrd 方法调用的简写。

表12-5：顺序比较操作符与PartialOrd方法

表　达　式	等价方法调用	默认定义
`x < y`	`x.lt(y)`	`x.partial_cmp(&y) == Some(Less)`
`x > y`	`x.gt(y)`	`x.partial_cmp(&y) == Some(Greater)`

`x <= y`	`x.le(y)`	`match x.partial_cmp(&y) {`
		`  Some(Less) | Some(Equal) => true,`
		`  _ => false,`
		`}`

`x >= y`	`x.ge(y)`	`match x.partial_cmp(&y) {`
		`  Some(Greater) | Some(Equal) => true,`
		`  _ => false,`
		`}`

如前面例子中所示，这里在调用相等方法时假设 `std::cmp::PartialOrd` 和 `std::cmp::Ordering` 在作用域中。

如果你知道两个类型的值之间总能分出个先后，就可以实现更严格的 `std::cmp::Ord` 特型：

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

这里的 `cmp` 方法只返回 `Ordering` 而非（像 `partial_cmp` 那样返回）`Opiton<Ordering>`。换句话说，`cmp` 要么返回相等，要么返回参数之间的相对顺序。几乎所有实现 `PartialOrd` 的类型也都会实现 `Ord`。在标准库中，`f32` 和 `f64` 是此规则的唯一例外。

由于复数没有自然顺序，因此没办法使用前几节中的 `Complex` 类型展示对 `PartialOrd` 的示例实现。为此，假设有以下类型，表示属于某个半开区间的数值集合：

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // 包含
    upper: T  // 不包含
}
```

我们想让这种类型的值实现部分排序，即如果一个区间完全落在另一个区间之前且没有重合，则该区间小于另一个区间。如果两个不相等的区间重合，即某一侧的某些元素小于另一侧的某些元素，则无法区分它们的顺序。两个相等的区间就是相等的。以下 `PartialOrd` 的实现满足以上规则：

```
use std::cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        if self == other { Some(Ordering::Equal) }
        else if self.lower >= other.upper { Some(Ordering::Greater) }
        else if self.upper <= other.lower { Some(Ordering::Less) }
        else { None }
    }
}
```

有了这个实现，就可以写出下面的代码进行测试了：

```
assert!(Interval { lower: 10, upper: 20 } <  Interval { lower: 20, upper: 40 });
assert!(Interval { lower: 7,  upper: 8  } >= Interval { lower: 0,  upper: 1  });
assert!(Interval { lower: 7,  upper: 8  } <= Interval { lower: 7,  upper: 8  });

// 重合的区间相互之间无法确定顺序
let left  = Interval { lower: 10, upper: 30 };
let right = Interval { lower: 20, upper: 40 };
assert!(!(left < right));
assert!(!(left >= right));
```

12.4 Index与IndexMut

通过实现 `std::ops::Index` 和 `std::ops::IndexMut` 特型，可以对相应类型使用类似 `a[i]` 这样的索引表达式。数组直接支持 `[]` 操作符，但对其他类型而言，表达式 `a[i]` 通常都是对 `*a.index(i)` 的简写，其中，`index` 是 `std::ops::Index` 特型的方法。不过，如果这个表达式被赋值或被可修改地借用，则 `a[i]` 就是对调用 `std::ops::IndexMut` 特型方法 `*a.index_mut(i)` 的简写。

以下是上述特型的定义：

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

注意，这两个特型以索引表达式的类型作为参数。为此，可以使用 `usize` 索引切片从而引用一个元素，因为切片实现了 `Index<usize>`。同样，可以使用表达式 `a[i..j]` 来引用子切片，因为切片也实现了 `Index<Range<usize>>`。这个表达式就是对以下方法调用的简写：

```
*a.index(std::ops::Range { start: i, end: j })
```

Rust 的 `HashMap` 和 `BTreeMap` 集合支持以任意可散列或可排序的类型作为索引。下列代码之所以有效，是因为 `HashMap<&str, i32>` 实现了 `Index<&str>`：

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("十", 10);
m.insert("百", 100);
m.insert("千", 1000);
m.insert("万", 1_0000);
m.insert("亿", 1_0000_0000);

assert_eq!(m["十"], 10);
assert_eq!(m["千"], 1000);
```

其中的索引表达式等价于：

```
use std::ops::Index;
assert_eq!(*m.index("十"), 10);
assert_eq!(*m.index("千"), 1000);
```

Index 特型的关联类型 Output 限定了索引表达式产出的类型，对 HashMap 而言，Index 实现的 Output 类型就是 i32。

IndexMut 特型扩展了 Index，增加了一个接收可修改的 self 引用作为参数的 index_mut 方法，该方法返回对一个 Output 值的可修改引用。当索引表达式出现在必要的上下文中的时候，Rust 会自动选择 index_mut。比如，假设有如下代码：

```
let mut desserts = vec!["Howalon".to_string(),
                        "Soan papdi".to_string()];
desserts[0].push_str(" (fictional)");
desserts[1].push_str(" (real)");
```

因为 push_str 方法操作的是 &mut self，所以上面最后两行代码等价于：

```
use std::ops::IndexMut;
(*desserts.index_mut(0)).push_str(" (fictional)");
(*desserts.index_mut(1)).push_str(" (real)");
```

IndexMut 有一个限制，就是根据设计它必须返回对某个值的可修改引用。这就是不能使用类似 m[" 十 "] = 10 这样的表达式给 HashMap m 插入值的原因。要实现这个插值操作，散列表需要先为 " 十 " 创建一个记录，以某个值作为其默认值，然后再返回对该记录的可修改引用。然而，并非所有类型都能以很小的代价使用默认值，有些默认值删除起来还是比较费事的。仅仅为了赋值就创建这么一个立即要删除的默认值是一种浪费。（Rust 后续版本有计划对此做出改进。）

索引最频繁的使用场景就是集合。比如，假设我们有一个位图图片，类似于第 2 章曼德布洛特集绘制程序中创建的那个。当时程序中包含如下代码：

```
pixels[row * bounds.0 + column] = ...;
```

假如有一个 Image<u8> 类型的二维数组，那就可以不必像这样写出数学计算表达式而直接访问像素了：

```
image[row][column] = ...;
```

为此，需要先声明一个结构体：

```
struct Image<P> {
    width: usize,
    pixels: Vec<P>
}

impl<P: Default + Copy> Image<P> {
    /// 创建给定大小的一张新图片
    fn new(width: usize, height: usize) -> Image<P> {
        Image {
            width,
```

```
            pixels: vec![P::default(); width * height]
        }
    }
}
```

而以下就是满足要求的 Index 和 IndexMut 的实现：

```
impl<P> std::ops::Index<usize> for Image<P> {
    type Output = [P];
    fn index(&self, row: usize) -> &[P] {
        let start = row * self.width;
        &self.pixels[start .. start + self.width]
    }
}

impl<P> std::ops::IndexMut<usize> for Image<P> {
    fn index_mut(&mut self, row: usize) -> &mut [P] {
        let start = row * self.width;
        &mut self.pixels[start .. start + self.width]
    }
}
```

这样，对 Image 的索引会返回一个像素切片，而对这个切片的索引会返回个别像素。

需要注意的是，在使用 image[row][column] 索引像素时，如果 row 越界，则 .index() 方法会尝试索引超出范围的 self.pixels，从而触发诧异。这也是 Index 和 IndexMut 实现应有的行为：越界访问会被发现并导致诧异，就跟索引数组、切片或向量时越界一样。

12.5 其他操作符

并不是所有的操作符都可以在 Rust 中重载。截止到 Rust 1.17，错误检查操作符 ? 只能用于 Result 值。类似地，逻辑操作符 && 和 || 也仅限于布尔值。.. 操作符只能用于创建 Range 值，& 操作符只能借用引用，而 = 操作符只能转移或复制值。上述这些操作符都不支持重载。

解引用操作符 *val 和访问字段及调用方法的点操作符（如 val.field 和 val.method()）可以使用 Deref 和 DerefMut 特型来重载，具体细节下一章会介绍。（不在本章介绍主要是因为这些特型不仅仅是重载几个操作符那么简单。）

Rust 不支持重载函数调用操作符（f(x)）。如果你需要一个可调用的值，通常写一个闭包就可以了。第 14 章在介绍 Fn、FnMut 和 FnOnce 等特殊的特型时将解释其中的工作原理。

第13章

实用特型

> 科学无非就是在自然界的多样性中寻找和发现一致性。或者更确切地讲，从我们经验的多样性中发现一致性。用 Coleridge 的话说，诗、画、艺术，同样是从多样性中发现一致性。
>
> ——Jacob Bronowski

除了操作符重载，也就是前一章介绍的内容，有些内置特型可以让我们直接修改 Rust 语言和标准库的一部分。

- 可以使用 Drop 特型在值超出作用域时清除它，类似 C++ 中的解构函数。
- 像 Box<T> 和 Rc<T> 这样的智能指针类型可以实现 Deref 特型，从而让指针反映封装值的方法。
- 通过实现 From<T> 和 Into<T> 特型，可以告诉 Rust 怎么将值从一种类型转换为另一种类型。

本章将集中介绍 Rust 标准库中的一批有用的特型，具体参见表 13-1。

表13-1：实用特型概述

特　型	简　介
Drop	解构函数。清除值时 Rust 自动运行的清除代码
Sized	标记特型，针对编译时可以知道大小的类型（而不是像切片那样动态大小的类型）
Clone	针对支持克隆值的类型
Copy	标记特型，针对可以简单地对内存中包含的值进行逐字节复制来克隆的类型
Deref 与 DerefMut	智能指针类型的特型
Default	针对有合理“默认值”的类型
AsRef 与 AsMut	转换特型，借用某种类型的引用

（续）

特　型	简　介
Borrow 与 BorrowMut	转换特型，类似 AsRef/AsMut，但额外保证一致的散列、顺序和相等
From 与 Into	转换特型，将某种类型的值转换为另一种类型
ToOwned	转换特型，将引用转换为所有值

当然，还有其他重要的标准库特型。比如，第 15 章将介绍的 Iterator 和 IntoIterator，第 16 章将介绍的 Hash 特型（针对计算散列码），还有第 19 章将介绍的用于标记线程安全类型的 Send 和 Sync。

13.1 Drop

当一个值的所有者离开时，Rust 会**清除**（drop）这个值。清除值涉及释放相关的值、堆存储空间，以及该值拥有的系统资源。清除会在各种条件下发生，包括变量超出作用域、表达式的值被；操作符丢弃、截断向量时从末尾删除其元素，等等。

很大程度上，Rust 会自动处理清除值。比如，假设定义了下面这个类型：

```
struct Appellation {
    name: String,
    nicknames: Vec<String>
}
```

Appellation 拥有字符串的内容和向量元素缓冲区对应的堆存储空间。Rust 会在 Appellation 被清除时自动释放这些存储空间，而无须在代码中额外声明。然而，如果你愿意，也可以通过实现 std::ops::Drop 特型自定义 Rust 清除你的类型值的方式：

```
trait Drop {
    fn drop(&mut self);
}
```

Drop 的实现类似于 C++ 中的解构函数或其他语言中的终结器（finalizer）。在清除值时，如果该值实现了 std::ops::Drop，那么 Rust 会在按常规清除其字段或元素拥有的值之前先调用它的 drop 方法。这种对 drop 方法的隐式调用是调用该方法的唯一方式。如果你显式地调用该方法，Rust 则会报错。

因为 Rust 在清除其字段或元素之前调用 Drop::drop，所以该方法接收到的值始终是完全初始化的。为 Appellation 类型实现 Drop 的代码可以任意使用其字段：

```
impl Drop for Appellation {
    fn drop(&mut self) {
        print!("Dropping {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (AKA {})", self.nicknames.join(", "));
        }
        println!("");
    }
}
```

基于以上实现，可以写出下面的代码：

```
{
    let mut a = Appellation { name: "Zeus".to_string(),
                              nicknames: vec!["cloud collector".to_string(),
                                              "king of the gods".to_string()] };

    println!("before assignment");
    a = Appellation { name: "Hera".to_string(), nicknames: vec![] };
    println!("at end of block");
}
```

在把第二个 Appellation 赋值给 a 时，第一个会被清除，而当离开 a 的作用域时，第二个会被清除。因此，以上代码会打印出如下结果：

```
before assignment
Dropping Zeus (AKA cloud collector, king of the gods)
at end of block
Dropping Hera
```

既然为 Appellation 实现的 std::ops::Drop 除了打印消息什么也没做，那它的内存到底是怎么释放的呢？ Vec 类型实现了 Drop，可以清除自己的每个元素，然后释放它们占用的堆内存空间。String 内部使用 Vec<u8> 保存其文本，因此 String 自身不需要实现 Drop，而是由 Vec 负责释放其字符。同样的原理也可以延伸到 Appellation 值：在一个值被清除后，最终其 Vec 对 Drop 的实现会负责释放每个字符串的内容，直到释放保存向量元素的缓存区。保存 Appellation 值自身的内存当然也有所有者，可能是一个局部变量或某个数据结构，它们会负责释放对应的内存。

如果变量的值被转移到了其他地方，导致变量在超出作用域时处于未初始化状态，Rust 则不会尝试清除该变量，因为这个变量里已经没有值需要清除了。

根据控制流的不同，无论变量的值是否被转移，这个原理都是适用的。在这种情况下，Rust 会用一个不可见的标志跟踪变量的状态，该标志表示变量的值是否需要被清除：

```
let p;
{
    let q = Appellation { name: "Cardamine hirsuta".to_string(),
                          nicknames: vec!["shotweed".to_string(),
                                          "bittercress".to_string()] };
    if complicated_condition() {
        p = q;
    }
}
println!("Sproing! What was that?");
```

根据 complicated_condition 条件返回 true 还是 false，Appellation 的值可能由 p 或 q 所拥有，而另一个会变成未初始化。这个值跑到哪里去，将决定它会在 println! 之前或之后被清除，因为 q 的作用域在 println! 之前结束，而 p 的作用域在 println! 之后结束。尽管一个值可以转移多次，但 Rust 对这个值只会清除一次。

如果不是自定义类型拥有 Rust 不知道的资源，一般不需要实现 std::ops::Drop。比如，在

Unix 系统上，Rust 标准库在内部使用以下类型表示操作系统的文件描述符：

```
struct FileDesc {
    fd: c_int,
}
```

这个 `FileDesc` 的 `fd` 字段只是文件描述符的编号，应该在程序结束时关闭。而 `c_int` 是 `i32` 的别名。标准库为 `FileDesc` 实现 `Drop` 的代码如下：

```
impl Drop for FileDesc {
    fn drop(&mut self) {
        let _ = unsafe { libc::close(self.fd) };
    }
}
```

这里的 `libc::close` 是 C 库的 `close` 函数在 Rust 中的名字。Rust 的代码只能在 `unsafe` 块中调用 C 函数，因此标准库就在这里使用了 `unsafe` 块。

如果某类型实现了 `Drop`，就不能再实现 `Copy` 特型。如果一个类型是 `Copy`，就意味着简单的字节对字节的复制已经足以产生一个值的独立副本。不过，在同一份数据上不止一次调用同一个 `drop` 方法通常是错误的。

标准前置模块中包含一个清除值的函数 `drop`，但是其定义一点也不新奇：

```
fn drop<T>(_x: T) { }
```

换句话说，它接收一个参数的值，从调用者那里取得所有权，然后什么也不做。Rust 会在超出作用域时清除 `_x` 的值，跟清除其他变量的值一样。

13.2 Sized

所谓**固定大小的类型**（sized type），指的是其值在内存中都具有相同大小。Rust 中几乎所有类型都是有大小的，比如每个 `u64` 占 8 字节，每个 `(f32, f32, f32)` 元组占 12 个字节。甚至枚举类型都是有大小的，无论最终表示哪种变体，一个枚举类型的值始终占能保存其最大变体的空间。虽然 `Vec<T>` 拥有分配在堆上的大小可变的缓冲区，但 `Vec` 值本身只包含一个指向该缓冲区的指针、缓冲区的容量及其长度。因此 `Vec<T>` 也是固定大小的类型。

不过，Rust 也有一些**非固定大小的类型**（unsized type），即其值的大小并不固定。比如，字符串切片类型 `str`（注意没有 `&`）是非固定大小的。字符串字面量 `"diminutive"` 和 `"big"` 是对占用 10 和 3 字节 `str` 切片的引用，如图 13-1 所示。像 `[T]`（同样没有 `&`）这样的数组切片类型也是非固定大小的，即共享引用 `&[u8]` 可以指向任意大小的 `[u8]` 切片。因为 `str` 和 `[T]` 类型表示大小不确定的值的集合，所以它们是非固定大小的类型。

Rust 中另一个常见的非固定大小类型是对特型目标的引用。正如 11.1.1 节所解释的，特型目标是一个指向实现了给定特型的某个值的指针。比如，类型 `&std::io::Write` 和 `Box<std::io::Write>` 都是指向实现了 `Write` 特型的某个值的指针。引用目标可能是一个文件、一个网络套接口，或者实现了 `Write` 的自定义类型。因为实现 `Write` 的类型是可以扩充的，所以 `Write` 作为类型被认为是非固定大小的，即其值的大小可变。

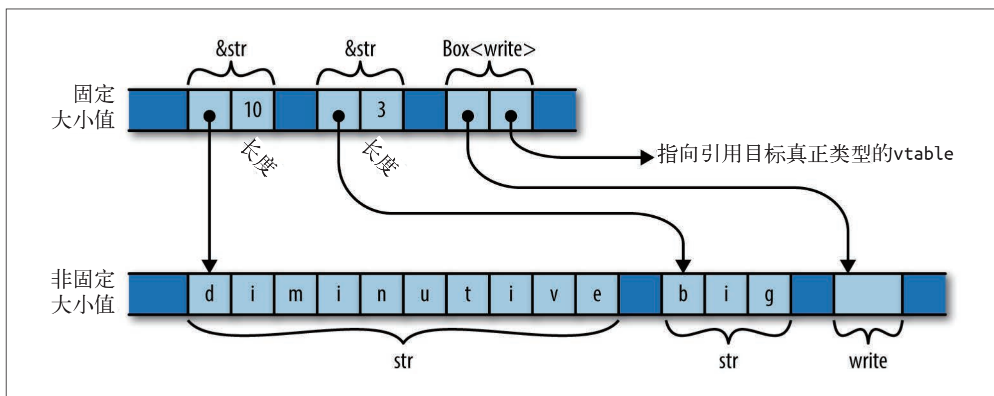


图 13-1：对非固定大小值的引用

Rust 不能在变量中存储非固定大小的值，也不能将它们作为参数传递。只能通过 `&str` 或 `Box<Write>` 这样本身是固定大小的指针来使用它们。如图 13-1 所示，指向非固定大小值的指针始终是**胖指针**，占两个字宽。胖指针既包含指向切片的指针，也包含切片的长度。而特型目标也包含一个指向方法实现的虚拟表的指针。

特型目标和切片的指针恰好具有对称性。对这两者而言，类型都缺少使用它们的必要信息。不知道长度，就不能使用索引访问 `[u8]` 值，而不知道对 `Write` 的特定实现，就不能调用 `Box<Write>` 上的方法。同样对这两者而言，胖指针补充了类型缺少的信息，给出了长度和虚拟表的指针。遗漏的静态信息被动态信息所取代。

所有固定大小的类型都实现了 `std::marker::Sized` 特型，这个特型没有方法或关联类型。Rust 为其适用的所有类型自动实现了这个特型，开发者不能自己实现。唯一需要使用 `Sized` 的场景，就是绑定类型变量。比如，`T: Sized` 绑定要求 `T` 必须是一个编译时大小已知的类型。这种特型称为**标记特型**（marker trait），因为 Rust 语言自身使用它们将某些类型标记为具有关注的特征。

由于非固定大小的类型具有很大局限性，因此大多数泛型变量应该被限制为使用 `Sized` 类型。事实上，正是因为太常见了，所以 Rust 隐式地将其作为了默认值。换句话说，如果你写 `struct S<T> { ... }`，那么 Rust 会将其理解为 `struct S<T: Sized> { ... }`。假如不想这样限制 `T`，就必须明确地写出来，比如写成 `struct S<T: ?Sized> { ... }`。这里的语法 `?Sized` 只能在这种情况下使用，其含义是“不一定是 `Sized`”。比如，如果你写的是 `struct S<T: ?Sized> { b: Box<T> }`，那么 Rust 会允许使用 `S<str>` 和 `S<Write>`（此时箱子变成了一个胖指针），除此之外，还允许使用 `S<i32>` 和 `S<String>`（此时箱子是一个普通指针）。

虽然有局限性，但非固定大小的类型让 Rust 的类型系统运行得更顺畅。阅读标准库文档的时候，你偶尔会发现类型变量上的 `?Sized` 绑定。此时通常都意味着给定类型只是指向，并且允许相关代码操作切片和特型目标以及普通的值。如果类型变量具有 `?Sized` 绑定，人们通常称其为**不知是否固定大小**（questionably sized），即可能是 `Sized`，也可能不是。

除了切片和特型目标，还有一种非固定大小类型。结构体类型的最后一个字段（仅最后一

个字段）可能是非固定大小的，而此时的结构体本身也是非固定大小的。比如，Rc<T> 引用计数指针在内部被实现为一个指向私有类型 RcBox<T> 的指针，该类型用于保存类型 T 及其引用计数。下面是 RcBox 的简化定义：

```
struct RcBox<T: ?Sized> {
    ref_count: usize,
    value: T,
}
```

这里 value 字段的值是 T，也就是要对它保存 Rc<T> 的引用计数。Rc<T> 解引用为一个对这个字段的指针。ref_count 字段保存引用计数。

可以在 RcBox 中使用固定大小的类型，如 RcBox<String>，结果就是一个固定大小的结构体类型。也可以在 RcBox 中使用非固定大小的类型，如 RcBox<std::fmt::Display>（其中 Display 是可以使用 println! 及类似的宏格式化的类型需要实现的特型），结果 RcBox<Display> 就是一个非固定大小的结构体类型。

不能直接构建 RcBox<Display> 值。相反，必须首先创建一个普通的固定大小的 RcBox，并让其 value 类型实现 Display，类似于 RcBox<String>。然后，Rust 允许你将引用 &RcBox<String> 转换为胖引用 &RcBox<Display>：

```
let boxed_lunch: RcBox<String> = RcBox {
    ref_count: 1,
    value: "lunch".to_string()
};

use std::fmt::Display;
let boxed_displayable: &RcBox<Display> = &boxed_lunch;
```

在给函数传值的时候会隐式地发生这种转换。因此，可以给接收 &RcBox<Display> 的函数传一个 &RcBox<String>：

```
fn display(boxed: &RcBox<Display>) {
    println!("For your enjoyment: {}", &boxed.value);
}

display(&boxed_lunch);
```

这会产生以下输出：

```
For your enjoyment: lunch
```

13.3 Clone

std::clone::Clone 特型适用于可以复制自身的类型。Clone 的定义如下：

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

clone 方法应该构建 self 的一个独立副本并返回它。因为这个方法的返回类型是 Self，而且函数不可能返回非固定大小的值，所以 Clone 特型本身扩展了 Sized 特型。这样就具有了将实现的 Self 类型绑定为 Sized 的效果。

克隆一个值通常涉及创建该值所拥有一切内容的副本及分配内存，因此 clone 无论在时间消耗还是内存占用方面都可能比较昂贵。例如，克隆 Vec<String> 不仅仅要复制向量，还要复制其包含的所有 String 元素。这就是 Rust 不自动克隆值，而是要求你明确地调用一个方法的原因。Rc<T> 和 Arc<T> 这样的引用计数指针属于例外，克隆它们只会简单地递增相应的引用计数，然后返回新指针。

clone_from 方法将 self 修改为 source 的一个副本。这个方法的默认定义只是简单地克隆了 source，然后将副本转移到 *self 中。这样是没有问题的，但对于某些类型而言，还有实现同样效果的更快方式。比如，假设 s 和 t 都是 String，则语句 s = t.clone(); 必须先克隆 t，清除 s 原来的值，然后再将克隆的值转移到 s 中。这涉及一次堆分配和一次堆释放。如果原始 s 值所有的堆缓冲区有足够的容量可以装下 t 的内容，就无须分配或者释放了：只要将 t 的文本复制到 s 的缓冲区，再调整一下长度即可。在泛型代码中，应该尽可能使用 clone_from，从而在可能的情况下应用这种优化。

如果你对 Clone 的实现只是简单地对自有类型的每个字段或元素应用 clone，然后再基于克隆的副本构建一个新值，那么 clone_from 的默认定义就足够用了。这种情况下，只要在类型定义上面加上 #[derive(Clone)]，Rust 就会为你实现它。

标准库中很多执行复制操作有意义的类型实现了 Clone。原始类型 bool 和 i32 实现了 Clone，容器类型 String、Vec<T> 和 HashMap 也实现了 Clone。有些类型实现复制操作没有意义，比如 std::sync::Mutex 就没有实现 Clone。有些类型可以复制，比如 std::fs::File，但在操作系统没有必要资源时复制会失败。这些类型也没有实现 Clone，因为 clone 必须万无一失。相反，std::fs::File 提供了一个 try_clone 方法，这个方法返回可以报告错误的 std::io::Result<File>。

13.4 Copy

第 4 章曾解释过，对于大多数类型而言，赋值会转移值，而不是复制值。转移值更有利于跟踪变量所拥有的资源。但 4.3 节中也指出了例外，即不拥有任何资源的简单类型可以是 Copy 类型，这种类型的赋值会生成值的副本，而不是转移值并让原始变量变成未初始化。

当时，我们并没有详细说明什么是 Copy，现在可以说了：如果类型实现了 std::marker::Copy 标记特型，那么它就是 Copy 类型。std::marker::Copy 标记特型的定义如下：

```
trait Copy: Clone { }
```

自定义类型实现它也很简单：

```
impl Copy for MyType { }
```

但由于 Copy 是一个标记特型，对语言有着特殊的意义，因此 Rust 只允许类型在字节对字节的深度复制能满足要求的情况下实现 Copy。那些可能拥有任意资源，比如堆缓冲区或操

作系统勾柄的类型，不能实现 Copy。

任何实现 Drop 特型的类型不能是 Copy。Rust 认为如果一个类型需要特殊的清理代码，那就一定需要特殊的复制代码，因此不能是 Copy。

与 Clone 一样，可以使用 #[derive(Copy)] 让 Rust 为你派生 Copy。事实上，通过 #[derive(Copy, Clone)] 同时派生这两者的情况是很常见的。

让类型实现 Copy 之前要想清楚。虽然实现 Copy 可以让类型更容易使用，但这样也给其实现带来了严重的限制。隐式复制的代价也可能是很大的。4.3 节曾详细解释过其中的利害。

13.5 Deref与DerefMut

通过实现 std::ops::Deref 和 std::ops::DerefMut 特型，可以修改解引用操作符 * 和 . 在自定义类型上的行为。Box<T> 和 Rc<T> 这样的指针类型实现了这两个特型，从而可以像 Rust 内置的指针类型一样行事。比如，如果你有一个 Box<Complex> 类型的值 b，那么 *b 引用的就是 b 指向的 Complex 值，而 b.re 引用其实数部分。如果是上下文赋值或从引用目标借用可修改的引用，那么 Rust 会使用 DerefMut（可变解引用）特型。否则，使用 Deref 取得只读的权限足矣。

这两个特型的定义如下：

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

deref 和 deref_mut 方法接收 &Self 引用并返回 &Self::Target 引用。Target 应该是 Self 包含、拥有或引用的资源。比如，对于 Box<Complex> 来说，Target 的类型就是 Complex。要注意，DerefMut 扩展了 Deref：如果可以解引用并修改资源，那么就应该可以借用一个对它的共享引用。因为这两个方法返回的引用具有与 &self 一样长的生命期，所以 self 会在返回引用的生命期内始终保持被借用。

Deref 和 DerefMut 特型也扮演了另一个角色。由于 deref 接收 &self 引用并返回 &Self::Target 引用，因此 Rust 会利用这一点自动将前一种类型的引用转换为后一种类型的引用。换句话说，如果插入一次 deref 调用可以防止类型错配，那 Rust 会为你插入一次。实现 DerefMut 可以实现对可修改引用的类型转换。这种类型转换称为**解引用强制转型**（deref coercion），即一种类型被“强制”表现出另一种类型的行为。

虽然解引用强制类型转型并非不能明确写出来，但它们很方便。

- 如果有一个 Rc<String> 值 r,你想对它调用 String::find,那么可以简单地写作 r.find('?')，而不需要写成 (*r).find('?')。这里的方法调用隐式借用了 r，而 &Rc<String> 被强制转换为 &String，因为 Rc<T> 实现了 Deref<Target=T>。

- 可以在 String 值上使用 split_at 等方法，即使 split_at 是 str 切片类型的方法，因为 String 实现了 Deref<Target=str>。这样，String 就不需要再实现 str 的所有方法了，因为可以将 &String 强制转型为 &str。
- 如果有一个字节向量 v，你想将它传给一个期待字节切片 &[u8] 的函数，那么可以将 &v 作为参数，因为 Vec<T> 实现了 Deref<Target=[T]>。

必要情况下，Rust 会连续多次应用解引用强制转型。比如，使用前面提到的强制转换，可以直接在 Rc<String> 上调用 split_at，因为 &Rc<String> 解引用为 &String，&String 又解引用为 &str，而 &str 有 split_at 方法。

比如，假设有以下类型：

```
struct Selector<T> {
    /// 此Selector中可用的元素
    elements: Vec<T>,

    /// elements中“当前”元素的索引。Selector类似于当前元素的指针
    current: usize
}
```

为了让 Selector 能够像文档注释中所说的那样，必须为这个类型实现 Deref 和 DerefMut：

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Selector<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.elements[self.current]
    }
}

impl<T> DerefMut for Selector<T> {
    fn deref_mut(&mut self) -> &mut T {
        &mut self.elements[self.current]
    }
}
```

基于以上实现，就可以像下面这样使用 Selector 了：

```
let mut s = Selector { elements: vec!['x', 'y', 'z'],
                       current: 2 };

// 因为Selector实现了Deref，所以可以使用*操作符引用它的当前元素
assert_eq!(*s, 'z');

// 通过解引用强制转型直接在Selector上使用char的方法断言'z'是一个字母
assert!(s.is_alphabetic());

// 通过给Selector的引用赋值，将'z'修改为'w'
*s = 'w';

assert_eq!(s.elements, ['x', 'y', 'w']);
```

Deref 和 DerefMut 特型的设计初衷是为了实现智能指针类型（如 Box、Rc 和 Arc），以及某

些会频繁通过引用来使用的类型的所有者版本（如 Vec<T> 和 String 就是 [T] 和 str 的所有者版本）。如果目的仅仅是让 Target 类型的方法自动在类型上可见（就像 C++ 中让基类的方法在子类上可见那样），则不应该实现 Deref 和 DerefMut。这样并不能保证始终有效，一旦出问题则很难发现。

解引用强制转型有一个问题可能导致困惑，即 Rust 通过应用这种机制来解决类型冲突，但不会应用它来满足类型变量的绑定。比如，下面的代码可以正常运行：

```
let s = Selector { elements: vec!["good", "bad", "ugly"],
                   current: 2 };

fn show_it(thing: &str) { println!("{}", thing); }
show_it(&s);
```

在调用 show_it(&s) 时，Rust 看到了一个类型为 &Selector<&str> 的实参和一个类型为 &str 的形参，并发现了 Deref<Target=str> 实现。于是，就把调用重写为了 show_it(s.deref())，而这正是我们想要的。

然而，如果把 show_it 改为泛型函数，Rust 转眼就变得不再合作了：

```
use std::fmt::Display;
fn show_it_generic<T: Display>(thing: T) { println!("{}", thing); }
show_it_generic(&s);
```

Rust 抱怨道：

```
error[E0277]: the trait bound `Selector<&str>: Display` is not satisfied
    |
542 |         show_it_generic(&s);
    |         ^^^^^^^^^^^^^^^ trait `Selector<&str>: Display` not satisfied
    |
```

这可能令人不解：为什么把函数改写成泛型函数就会报错呢？没错，Selector<&str> 并未实现 Display，但它解引用为 &str，后者则实现了 Display。

由于我们传递的实参类型是 &Selector<&str>，而函数的形参类型是 &T，因此类型变量 T 必须是 Selector<&str>。然后，Rust 检查绑定 T: Display 是否满足。因为它不会在满足类型变量的绑定时应用解引用强制转型，所以检查失败。

要解决这个问题，可以使用 as 操作符显式进行强制转换：

```
show_it_generic(&s as &str);
```

13.6 Default

某些类型拥有明显合理的默认值，比如默认向量或字符串是空的、默认数值是 0、默认 Option 是 None，等等。这些有默认值的类型都实现了 std::default::Default 特型：

```
trait Default {
    fn default() -> Self;
}
```

default 方法简单地返回类型 Self 的一个新值。String 对 Default 的实现很直观：

```
impl Default for String {
    fn default() -> String {
       String::new()
    }
}
```

Rust 的所有集合类型，比如 Vec、HashMap、BinaryHeap 等，都实现了 Default，而且 default 方法都返回空集合。这在你想构建某个值的集合，但希望让调用者决定到底构建什么集合时非常有用。例如，Iterator 特型的 partition 方法会将迭代器产生的值分成两个集合，而哪个值分到哪个集合由一个闭包来决定：

```
use std::collections::HashSet;
let squares = [4, 9, 16, 25, 36, 49, 64];
let (powers_of_two, impure): (HashSet<i32>, HashSet<i32>)
    = squares.iter().partition(|&n| n & (n-1) == 0);

assert_eq!(powers_of_two.len(), 3);
assert_eq!(impure.len(), 4);
```

闭包 |&n| n * (n-1) == 0 利用某些位操作来识别 2 的幂次方的数值，而 partition 利用它生成了两个 HashSet。当然，partition 并非只能生成 HashSet，可以让它生成你想要的任意集合类型。只要该集合类型实现了 Default，默认生成一个空集合，以及 Extend<T>，能够向集合中添加 T 即可。String 实现了 Default 和 Extend<char>，因此你的代码可以这样写：

```
let (upper, lower): (String, String)
    = "Great Teacher Onizuka".chars().partition(|&c| c.is_uppercase());
assert_eq!(upper, "GTO");
assert_eq!(lower, "reat eacher nizuka");
```

Default 的另一个常见用途是为表示大量参数集合（大部分参数通常不需要改变）的结构体生成默认值。例如，glium 包为强大而复杂的 OpenGL 图形库提供 Rust 绑定。glium::DrawParameters 结构体包含 22 个字段，每个字段都用于控制 OpenGL 应该如何渲染某些图形的位等细节。glium 的 draw 函数期待一个 DrawParameters 结构体作为参数。由于 DrawParameters 实现了 Default，因此可以创建一个传给 draw，只要明确给出想要改变的字段即可：

```
let params = glium::DrawParameters {
    line_width: Some(0.02),
    point_size: Some(0.02),
    .. Default::default()
};

target.draw(..., &params).unwrap();
```

这里调用 Default::default() 是为了创建以所有字段的默认值初始化的 DrawParameters 结构体，然后使用结构体的 .. 语法为需要修改的 line_width 和 point_size 字段创建新的值，从而为传给 target.draw 做好准备。

如果类型 T 实现了 Default，那么标准库会自动为 Rc<T>、Arc<T>、Box<T>、Cell<T>、

RefCell<T>、Cow<T>、Mutex<T> 和 RwLock<T> 实现 Default。说到默认值，以 Rc<T> 为例，其表示一个指向类型 T 默认值的 Rc。

如果元组类型的所有元素类型都实现了 Default，且该元组类型也实现了 Default，那么这个元组默认会持有每个元素的默认值。

Rust 没有为结构体类型隐式实现 Default，但是如果结构体的所有字段都实现了 Default，则可以使用 #[derive(Default)] 自动为结构体实现 Default。

任何 Option<T> 的默认值都是 None。

13.7 AsRef与AsMut

一个类型如果实现了 AsRef<T>，就意味着可以有效地向它借用一个 &T。顾名思义，AsMut 就是可修改引用。这两个特型的定义如下：

```
trait AsRef<T: ?Sized> {
    fn as_ref(&self) -> &T;
}

trait AsMut<T: ?Sized> {
    fn as_mut(&mut self) -> &mut T;
}
```

比如说，Vec<T> 实现了 AsRef<[T]>，String 实现了 AsRef<str>。之所以可以将 String 的内容借用为字节数组，就是因为 String 也实现了 AsRef<[u8]>。

AsRef 通常用于使函数在接收参数的类型中更加灵活。例如，std::fs::File::open 函数的声明是这样的：

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

open 真正想要的是 &Path，即表示文件系统路径的类型。但从这个签名来看，open 接收任何可以从中借用 &Path 的类型，也就是任何实现了 AsRef<Path> 的类型。符合这个要求的类型包括 String 和 str、操作系统接口字符串类型 OsString 和 OsStr，当然还有 PathBuf 和 Path。完整的列表可以参考标准库文档。因此，可以给 open 传入一个字符串字面量：

```
let dot_emacs = std::fs::File::open("/home/jimb/.emacs")?;
```

标准库中所有的文件系统访问函数都可以像这样接收路径参数。对调用者而言，效果类似于 C++ 中重载的函数，只不过 Rust 采用的是另一种不同的手段，即规定函数可以接收哪些类型的参数。

当然，还需要进一步解释。字符串字面量是 &str，但实现 AsRef<Path> 的是 str，没有 &。正如 13.5 节所解释的，Rust 不会尝试解引用强制转型去满足类型变量绑定，因此这里不会发生强制转型。

所幸的是，标准库包含了一个非限制性（blanket）的实现：

```
impl<'a, T, U> AsRef<U> for &'a T
    where T: AsRef<U>,
```

```
        T: ?Sized, U: ?Sized
{
    fn as_ref(&self) -> &U {
        (*self).as_ref()
    }
}
```

换句话说，对于任意类型 T 和 U，如果 T: AsRef<U>，则 &T: AsRef<U> 也成立，即简单地跟随引用，然后一如既往。特别地，由于 str: AsRef<Path>，因此 &str: AsRef<Path> 也成立。某种意义上讲，这可以看成检查 AsRef 在类型变量上的绑定时的一种有限制的解引用强制转型。

基本上可以这样假定，如果一个类型实现了 AsRef<T>，那它也应该实现 AsMut<T>。不过，有些情况下这样并不合适。比如，前面曾提到过 String 实现了 AsRef<[u8]>，这没问题，因为每个 String 都对应着一个字节缓冲区，而从这个缓冲区中读取二进制数据是有意义的。可是，String 进一步保证其包含的字节是格式良好的 UTF-8 编码的 Unicode 文本。如果 String 实现了 AsMut<[u8]>，就相当于允许调用者随意修改 String 的字节，结果可能是无法再信任 String 是格式良好的 UTF-8 了。因此，一个类型是否有必要实现 AsMut<T>，要看修改了 T 之后是否会影响该类型的不变性约束。

虽然 AsRef 和 AsMut 都很简单，但它们作为标准化的泛型特型可以避免出现更具体的转型特型。如果可以实现 AsRef<Foo>，就不要定义自己的 AsFoo 特型。

13.8 Borrow与BorrowMut

std::borrow::Borrow 特型与 AsRef 类似：如果一个类型实现了 Borrow<T>，那么它的 borrow 方法可以从自身有效地借用一个 &T。不过，Borrow 增加了更多限制：只有当 &T 与它所借用的值具有同样的散列和比较特性时，一个类型才可以实现 Borrow<T>。（Rust 不会强制这一点，只是在文档中这样规定了这个特型的意图。）这使得在处理散列表或树中的键，或者处理由于其他原因将被散列或比较的值时，Borrow 都很有用。

在从 String 借用值时，这种区别很重要。比如，String 实现了 AsRef<str>、AsRef<[u8]> 和 AsRef<Path>，但这 3 种目标类型通常会有不同的散列值。只有 &str 切片能保证与对应的 String 有一样的散列化结果，因此 String 只实现了 Borrow<str>。

Borrow 的定义跟 AsRef 几乎一样，只是改了个名字：

```
trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}
```

Borrow 的用意是针对泛型散列表及其他关联集合类型的特殊情形。比如，假设有一个 std::collections::HashMap<String, i32> 将字符串映射到数值。这个表的键是 String，每条记录都有一个键。那么从这个表中查询一条记录的方法的签名应该长什么样？下面是一种方案：

```
impl HashMap<K, V> where K: Eq + Hash
{
```

```
    fn get(&self, key: K) -> Option<&V> { ... }
}
```

思路没问题，要查询记录，必须提供一个适当类型的匹配表的键。但在这里，K 是 String，即这个签名会强制每次调用 get 都传入一个 String 值。这显然很浪费。实际上这里只需要一个键的引用：

```
impl HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: &K) -> Option<&V> { ... }
}
```

稍微好一点了，但现在传入的键必须是 &String。如果你想查询一个常量字符串，必须得这样写：

```
hashtable.get(&"twenty-two".to_string())
```

这很荒谬：先在堆内存上分配一个 String 缓冲区，再把文本复制进去，接着用 &String 来借用它，然后传给 get，最后把它清理掉。

其实只要传入的值能产生跟键的类型可比较的散列值就足够了。&str 应该完全符合这个条件。因此，下面就是最终版本，也是标准库的实现：

```
impl HashMap<K, V> where K: Eq + Hash
{
    fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
        where K: Borrow<Q>,
              Q: Eq + Hash
    { ... }
}
```

换句话说，如果可以通过 &Q 借用记录的键，而且引用的散列和比较与键本身一样，那么显然 &Q 应该是可以接受的键类型。因为 String 实现了 Borrow<str> 和 Borrow<String>，所以这个最终版本的 get 允许按需传入 &String 或 &str 作为键。

Vec<T> 和 [T: N] 实现了 Borrow<[T]>。所有类字符串类型都允许借用其对应的切片类型：String 实现了 Borrow<str>、PathBuf 实现了 Borrow<Path>，等等。标准库中所有关联集合类型都使用 Borrow 来确定什么类型可以传给它们的查询函数。

标准库中包含一个非限制性实现，以便每个类型 T 都可以从自身借用：T: Borrow<T>。这样就确保了 &K 始终是在 HashMap<K, V> 中查询记录时可接受的类型。

为方便起见，所有 &mut T 类型也都实现了 Borrow<T>，跟之前一样返回共享引用 &T。这样就可以给集合查询函数传入一个可修改引用，而无须再重新借用一个共享引用，同时也模拟了 Rust 从可修改引用到共享引用的隐式转型。

BorrowMut 特型是 Borrow 针对可修改引用的版本：

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
```

前面介绍的实现 Borrow 所要具备的条件同样适用于 BorrowMut。

13.9 From与Into

std::convert::From 和 std::convert::Into 特型表示类型转换，即消费一种类型的值，然后返回另一种类型的值。AsRef 和 AsMut 特型是从一种类型借用另一种类型的引用，From 和 Into 则是取得它们参数的所有权，转换类型，再把结果的所有权返回给调用者。

这两个特型的定义是对称的：

```
trait Into<T>: Sized {
    fn into(self) -> T;
}

trait From<T>: Sized {
    fn from(T) -> Self;
}
```

标准库自动实现了每种类型到自身的简单转换，即每种类型 T 都实现了 From<T> 和 Into<T>。

虽然这两个特型只提供了做同一件事的两种方式，但它们因此也具有了不同的用途。

通常，可以使用 Into 让函数更灵活地接收参数。比如，如果这样写代码：

```
use std::net::Ipv4Addr;
fn ping<A>(address: A) -> std::io::Result<bool>
    where A: Into<Ipv4Addr>
{
    let ipv4_address = address.into();
    ...
}
```

那么 ping 函数不仅可以接收 Ipv4Addr 作为参数，也可以接收 u32 或 [u8; 4] 数组，因为后两个类型为方便起见也都实现了 Into<Ipv4Addr>。（有时候把 IPv4 地址当成一个 32 位值或者一个 4 字节数组来处理会更有用。）因为 ping 只知道 address 实现了 Into<Ipv4Addr>，所以在调用 into 时也不用指定你想要什么类型。只有一种类型可能有用，所以类型推断会帮你把这事给干了。

与前面介绍的 AsRef 一样，这里的效果非常像在 C++ 中重载了一个函数。基于前面的 ping 函数定义，可以像下面这样以多种类型调用它：

```
println!("{:?}", ping(Ipv4Addr::new(23, 21, 68, 141)));   // 传入Ipv4Addr
println!("{:?}", ping([66, 146, 219, 98]));               // 传入[u8; 4]
println!("{:?}", ping(0xd076eb94_u32));                   // 传入u32
```

不过，From 特型的角色却不一样，其 from 方法就像一个泛型构造函数，可以基于其他值产生一个当前类型的实例。比如，与 Ipv4Addr 有 from_array 和 from_u32 两个方法不同，From 只实现了 From<[u8;4]> 和 From<u32>，于是才允许这样写：

```
let addr1 = Ipv4Addr::from([66, 146, 219, 98]);
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

这也就是说，可以让类型推断帮我们整理要使用的实现。

有了 From 的适当实现，标准库可以自动实现对应的 Into 特型。在定义自己的类型时，如果它有只接收一个参数的构造函数，那应该将它们写成针对适当类型的 From<T> 的实现，然后就能自动获得对应的 Into 实现。

因为 from 和 into 转换方法取得它们参数的所有权，所以转换过程中可以使用原始值的资源来构建转换后的值。比如，假设有如下代码：

```
let text = "Beautiful Soup".to_string();
let bytes: Vec<u8> = text.into();
```

这里，String 对 Into<Vec<u8>> 的实现只需找到 String 的堆缓冲区并重新调整用途，无须更改，只是将其作为向量元素的缓冲区返回。这种转换无须分配或复制文本。这也是转移可以提高实现效率的另一个例子。

以上转换也是将较受限类型值放宽为更灵活类型值的一种不错的方式，不会影响受限类型的保证。比如，String 保证其内容始终是有效的 UTF-8，其修改方法通过谨慎的限制确保用户不会引入错误的 UTF-8。不过这个例子有效地将一个 String“降级”为了一个纯字节块，我们可以对它进行任意操作。比如，可以压缩它，或者将它与其他非 UTF-8 二进制数据组合。因为 into 取得的是参数的值，所以 text 在转换之后就变成未初始化状态了。这意味着我们可以自由访问之前 String 的缓冲区，而不会破坏任何现有的 String。

不过，低开销转换并非 Into 和 From 的承诺。尽管 AsRef 和 AsMut 转换预计开销不大，但 From 和 Into 转换可以分配、复制或处理值的内容。比如，String 实现了 From <&str>，可以将字符串切片复制到一个在堆内存上新分配的缓冲区并返回 String。而 std::collections::BinaryHeap<T> 实现了 From<Vec<T>>，它可以根据其算法的需要对元素进行比较和重新排序。

要注意的是，From 和 Into 仅限于永不失败的转换。单从方法类型参数的签名，根本看不出来某个转换会失败。为了让自定义类型在向内或向外转换时处理可能的失败，最好让函数或方法返回 Result 类型。

在 From 和 Into 被添加到标准库之前，Rust 代码中到处可见临时的转换特型和构造方法，每种类型都有一套。From 和 Into 把简化类型使用的惯例固定了下来，不仅实现者可以遵循，而且类型的用户也习以为常。

13.10 ToOwned

给定一个引用，产生其引用目标的所有型副本的通常方式是调用 clone 方法，当然前提是这个类型实现了 std::clone::Clone。但是，如果你想克隆一个 &str 或 &[i32] 怎么办？你想要的可能是一个 String 或 Vec<i32>，但这是 Clone 的定义所不允许的。按照定义，克隆一个 &T 必须返回一个类型 T 的值，而 str 和 [u8] 都是非固定大小的。因此它们甚至都不是函数可以返回的类型。

std::borrow::ToOwned 特型针对这个问题提供了稍微宽松一点的方案，可以把引用转换为所有型的值：

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;
}
```

与 clone 必须返回 Self 不同，to_owned 可以返回能够借用为 &Self 的任何类型，Owned 类型必须实现 Borrow<Self>。可以从 Vec<T> 借用一个 &[T]，因此 [T] 可以实现 ToOwned<Owned=Vec<T>>，只要 T 实现 Clone 即可，这样就可以把切片的元素复制到向量中。类似地，str 实现了 ToOwned<Owned=String>、Path 实现了 ToOwned<Owned=PathBuf>，等等。

13.11 Borrow与ToOwned实例：谦逊的奶牛（Cow）

要想用好 Rust，必须把与所有权相关的问题想清楚，比如一个函数到底应该按引用还是按值接收参数。通常，可以选择其中一种方式，而参数的类型反映了你的决定。但在某些情况下，只能到程序运行时才能决定到底是借用还是取得所有权更合适。std::borrow::Cow（clone on write，写时克隆）类型为此提供了一种解决方案。

std::borrow::Cow 的定义如下：

```
enum Cow<'a, B: ?Sized + 'a>
    where B: ToOwned
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Cow<B> 可以借用对 B 的一个共享引用，也可以拥有一个值，然后再借用这样一个引用。因为 Cow 实现了 Deref，所以可以把它当成一个对 B 的共享引用，进而调用任何方法。如果它是 Owned，就借用对这个所有值的共享引用；如果它是 Borrowed，就交出它持有的引用。

调用 to_mut 方法也可以取得对一个 Cow 值的可修改引用，此时方法返回的是 &mut B。如果 Cow 恰好是 Cow::Borrowed，to_mut 就直接调用引用的 to_owned 方法，取得对引用目标的所有型副本，将 Cow 改为 Cow::Owned，并借用对这个新所有值的可修改引用。这就是“写时克隆”。

类似地，Cow 的 into_owned 方法在必要时会将引用提升为所有型的值，然后返回它，将所有权转移给调用者，并在此过程中消费 Cow。

Cow 的一个常见用途是返回静态分配的字符串常量或者计算的字符串。假设你需要将一个错误枚举转换为一条消息。大多数变体可以用固定大小的字符串来处理，但有些也需要在消息中包含额外的数据。可以返回一个 Cow<'static, str>：

```
use std::path::PathBuf;
use std::borrow::Cow;
fn describe(error: &Error) -> Cow<'static, str> {
    match *error {
        Error::OutOfMemory => "out of memory".into(),
        Error::StackOverflow => "stack overflow".into(),
        Error::MachineOnFire => "machine on fire".into(),
        Error::Unfathomable => "machine bewildered".into(),
```

```
            Error::FileNotFound(ref path) => {
                format!("file not found: {}", path.display()).into()
            }
        }
    }
```

代码中使用了 Cow 对 Into 的实现来构建想要的值。match 语句的大多数分支返回一个 Cow::Borrowed，引用一个静态分配的字符串。但对于 FileNotFound 变体，则使用 format! 构建了一个包含给定文件名的消息。这个 match 语句的分支产生的是一个 Cow::Owned 值。

describe 的调用者如果不需要修改返回值，可以把 Cow 当成一个 &str：

```
println!("Disaster has struck: {}", describe(&error));
```

而需要一个所有型值的调用者也可以很容易得到它：

```
let mut log: Vec<String> = Vec::new();
...
log.push(describe(&error).into_owned());
```

在 Cow 的帮助下，describe 及其调用者得以把内存分配推迟到了真正必要的时候。

第 14 章

闭包

> 拯救环境！今天就创建闭包！
>
> ——Cormac Flanagan

排序整数向量很简单：

```
integers.sort();
```

然而现实当中，当我们想要排序某些数据时，这些数据往往不是整数向量。通常，我们会获取某种记录，但不能使用其内置的 `sort` 方法：

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort(); // 错误：根据什么进行排序？
}
```

Rust 抱怨说 `City` 没有实现 `std::cmp::Ord`。为此，需要指定按什么顺序来排序，比如：

```
/// 按人口排序城市的辅助函数
fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(city_population_descending); // 没问题了
}
```

这里的辅助函数 `city_population_descending` 接收 `City` 记录，然后提取出它的**键**（key），之后要通过这个键对数据进行排序。（这里返回负值是因为 `sort` 会增序排列数值，而我们想要的是减序：人口最多的城市排前面。）`sort_by_key` 方法将这个键函数作为参数。

这样可以达到目的。但编写这个辅助函数的更简洁方式是把它的逻辑写成**闭包**，即一个匿名函数表达式：

```
fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(|city| -city.population);
}
```

这里的 `|city| -city.population` 就是闭包。它接收一个参数 `city`，返回 `-city.population`。Rust 会根据使用闭包的上下文推断参数类型和返回类型。

标准库中还有一些特性也接收闭包。

- `Iterator` 的 `map` 和 `filter` 方法，用于操作顺序数据，第 15 章会介绍这些方法。
- 启动新系统线程的 `thread::spawn` 等线程 API。并发的核心是在线程间交换工作，而闭包可以方便地表示工作单元。第 19 章会介绍这些特性。
- 某些方法会在必要时计算默认值，比如 `HashMap` 的 `or_insert_with` 方法。这个方法会从 `HashMap` 中获取一个值或者创建一个值，用于计算默认值比较耗时的情形。默认值以闭包形式传入，但只会在必须创建新值时调用。

当然，如今匿名函数已经非常常见了，就连最初本来没有这个特性的 Java、C#、Python 和 C++ 也都支持了。接下来，假设我们已经了解了匿名函数，而专注于解释 Rust 的闭包有什么不同。本章会介绍如何在标准库方法中使用闭包、闭包如何在其作用域中“捕获”变量、如何编写自己的以闭包为参数的函数和方法，以及存储闭包以便将来用作回调。总之，本章会解释 Rust 闭包的工作原理，以及为什么它的速度超出想象。

14.1 捕获变量

闭包可以使用属于包含函数的数据。例如：

```
/// 按几个不同的统计指标排序
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

这里的闭包使用了 `stat`，而包含函数 `sort_by_statistic` 拥有这个变量。对此，我们说闭包“捕获”了 `stat`。这是闭包的几个经典特性之一，Rust 自然也支持。但在 Rust 中，这个特性会受到一些限制。

在大多数支持闭包的语言中，垃圾回收扮演着重要角色。例如下面这段 JavaScript 代码：

```
// 启动重排城市表格行的动画
function startSortingAnimation(cities, stat) {
    // 用于排序表格的辅助函数
    // 注意，这个函数引用了stat
    function keyfn(city) {
        return city.get_statistic(stat);
```

```
    }

    if (pendingSort)
        pendingSort.cancel();

    // 现在启动动画，传入keyfn
    // 后面排序算法会调用keyfn
    pendingSort = new SortingAnimation(cities, keyfn);
}
```

这里闭包 keyfn 保存在了新的 SortingAnimation 对象中，并会在 startSortingAnimation 返回后被调用。正常情况下，一个函数返回后，其所有变量和参数都会因超出作用域而被销毁。但在这里，JavaScript 引擎必须保留 stat，因为闭包引用了它。大多数 JavaScript 引擎会把 stat 分配到堆上，然后交给垃圾回收程序以后再回收。

Rust 没有垃圾回收，那如何实现这个特性呢？为回答这个问题，先来看两个例子。

14.1.1 借用值的闭包

首先，再来看看本节开头的例子：

```
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

这时候，Rust 在创建闭包时会自动借用对 stat 的引用。这合情合理：闭包引用了 stat，因此必须有一个对它的引用。

剩下的就简单了。闭包也遵守第 5 章描述的借用和生命期规则。特别地，因为闭包包含对 stat 的引用，所以 Rust 不会让它的存活期超过 stat。闭包只在排序期间使用，因此这个例子没什么问题。

简单来说，Rust 使用生命期而不是垃圾回收确保代码安全，但 Rust 的方式更快。即使是快速的 GC 分配也会比 Rust 把 stat 保存在栈上慢。

14.1.2 盗用值的闭包

第二个例子就没那么简单了：

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

这跟前面 JavaScript 的例子有点像：thread::spawn 接收一个闭包，并在新的系统线程中调用这个闭包。注意，|| 表示闭包没有参数。

新线程与调用程序并行运行。闭包返回后，新线程退出。（闭包的返回值会包装在 JoinHandle 中返回给调用线程。第 19 章会介绍 JoinHandle。）

同样，闭包 key_fn 包含对 stat 的引用。但这一次，Rust 不能保证安全使用。为此，它会拒绝这个程序：

```
error[E0373]: closure may outlive the current function, but it borrows `stat`,
              which is owned by the current function
  --> closures_sort_thread.rs:33:18
   |
33 | let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };
   |              ^^^^^^^^^^^^^^^^^^^^                       ^^^^
   |              |                                       `stat` is borrowed here
   |              may outlive borrowed value `stat`
```

实际上这里有两个问题，因为 cities 也属于不安全的共享。很简单，thread::spawn 创建的新线程不能保证自己在 cities 和 stat 被销毁（函数结束）前完成任务。

这两个问题的解决方案相同：告诉 Rust 把 cities 和 stat **转移**到使用它们的闭包中，而不要再引用它们。

```
fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = move |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

这里唯一的改动就是在两个闭包前面各自加了一个 move 关键字。move 关键字告诉 Rust，这个闭包不是在借用它使用的变量，而是要把它偷走。

第一个闭包 key_fn 获得了 stat 的所有权。而第二个闭包获得了 cities 和 key_fn 的所有权。

因此，Rust 为闭包提供了两种从包含函数取得数据的方式：转移和借用。实际上到这儿也就没什么可说的了，闭包也遵循第 4 章和第 5 章已经介绍过的转移和借用规则。下面稍微分析一下。

- 正如这门语言中的其他地方一样，如果闭包转移了一个可复制的值（如 i32），那它会复制该值。因此，如果 Statistic 恰好是一个可复制类型，那么在创建使用它的转移闭包之后，照样还可以使用 stat。
- 像 Vec<City> 这样的不可复制类型值才会真正转移。上面的代码通过转移闭包把 cities 转移到了新线程中。在创建这个闭包的代码后面，Rust 不会再让我们访问 cities。

- 巧合的是，上面的代码在闭包转移 cities 之后也不需要使用 cities 了。假如还需要使用 cities，解决办法也很简单。可以告诉 Rust 克隆 cities 并将副本保存到另一个变量中。闭包只能偷走它自己引用的那个副本。

在接受 Rust 严格的规则之后，我们也换来了重要的收益：线程安全。这完全归功于向量被转移了，而不是在多个线程间共享。如果是共享的，那么在新线程修改它时，老线程也不会放手。

14.2 函数与闭包类型

本章中，函数和闭包被当作值来使用。自然，这意味着它们也有自己的类型。例如：

```
fn city_population_descending(city: &City) -> i64 {
    -city.population
}
```

这个函数接收一个参数（&City），返回一个 i64 值。因此它的类型就是 fn(&City) -> i64。

可以像使用其他任何值一样使用函数。可以把函数保存在变量里。可以使用任何常用的 Rust 语法计算函数值：

```
let my_key_fn: fn(&City) -> i64 =
    if user.prefs.by_population {
        city_population_descending
    } else {
        city_monster_attack_risk_descending
    };

cities.sort_by_key(my_key_fn);
```

结构体可以包含函数类型的字段。泛型类型如 Vec 可以存储一批函数，只要它们的 fn 类型一样就行。函数值很小，一个 fn 值就是这个函数机器码的内存地址，与 C++ 中的函数指针类似。

一个函数可以接收另一个函数作为参数。例如：

```
/// 传入一组城市和一个测试函数，
/// 返回满足条件的所有城市
fn count_selected_cities(cities: &Vec<City>,
                         test_fn: fn(&City) -> bool) -> usize
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}

/// 一个测试函数的例子。注意，这个函数的类型是
/// fn(&City) -> bool，跟count_selected_cities
```

```
/// 函数的参数test_fn一致
fn has_monster_attacks(city: &City) -> bool {
    city.monster_attack_risk > 0.0
}

// 看一下多少城市有被怪物袭击的风险？
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

如果你熟悉 C/C++ 中的函数指针，就会发现 Rust 的函数值实际上就是指针。

虽然如此，听到闭包居然与函数**不是**同一种类型你一定会惊讶：

```
let limit = preferences.acceptable_monster_risk();
let n = count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // 错误：类型不匹配
```

第二个参数会导致类型错误。为支持闭包，必须修改函数的类型签名。修改后应该类似这样：

```
fn count_selected_cities<F>(cities: &Vec<City>, test_fn: F) -> usize
    where F: Fn(&City) -> bool
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}
```

这里只改了 `count_selected_cities` 的类型签名，没有改函数体。新版本是一个泛型函数，接收一个任意类型 `F` 的参数 `test_fn`，而 `F` 必须实现特型 `Fn(&City) -> bool`。所有以一个 `&City` 为参数且返回布尔值的函数和闭包都会自动实现这个特型。

```
fn(&City) -> bool   // fn类型（仅函数）
Fn(&City) -> bool   // Fn特型（包括函数和闭包）
```

这个特殊语法内置在这门语言中。中间的 `->` 和后面的返回值类型是可选的。如果省略，则返回值类型为 `()`。

新版本的 `count_selected_cities` 可以接收函数，也可以接收闭包：

```
count_selected_cities(
    &my_cities,
    has_monster_attacks); // 没问题

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // 同样没问题
```

为什么第一次尝试不行呢？因为闭包可以调用，但它不是 `fn`。闭包 `|city| city.monster_attack_risk > limit` 有自己的类型，但并不是 `fn`。

事实上，你写的每个闭包都有自己的类型，因为闭包可能包含（从包含作用域中借来或偷

来的）数据。这可能是任意多个变量，这些变量也可能是任何类型。因此编译器会为每个闭包创建一个临时类型，大到足以存储它的数据。任何两个闭包的类型都不相同。但所有闭包都会实现 `Fn` 特型，例子中的闭包就实现了 `Fn(&City) -> bool`。

由于每个闭包都有自己的类型，因此使用闭包的代码通常需要是泛型的，像 `count_selected_cities` 一样。每次都要写出泛型类型显得有点麻烦，下一节会解释这样设计的好处。

14.3 闭包的性能

Rust 闭包的设计保证它非常快，比函数指针还要快，快到完全可以用在强度和性能要求极高的环境下。如果你熟悉 C++ 的 lambda 表达式，就会发现 Rust 闭包跟它一样快且简洁，但更安全。

在大多数语言中，闭包是分配在堆上，动态分派，然后由垃圾回收程序负责回收的。因此创建、调用和回收它们都会多花那么一点点 CPU 时间。更麻烦的是，编译器很难对闭包应用**行内化**优化策略以减少函数调用并进而应用其他优化。总之，这些语言中的闭包可以慢到需要你考虑把它们从紧凑的内部循环（tight inner loop）中移出来。

Rust 闭包完全没有这些性能上的缺点。Rust 就没有垃圾回收。与 Rust 中的其他特性一样，闭包不会被分配到堆上，除非你把它们装到 `Box`、`Vec` 或其他容器里。而且因为每个闭包都有一个不同的类型，所以 Rust 编译器只要知道了你所调用闭包的类型，就可以将该闭包的代码行内化。这样在紧凑循环里使用闭包就没什么问题了，Rust 程序经常热衷于这样做，正如第 15 章将展示的那样。

图 14-1 展示了 Rust 闭包在内存中的布局。图的上方是闭包会引用的两个局部变量：字符串 `food` 和值为 27 的简单枚举 `weather`。

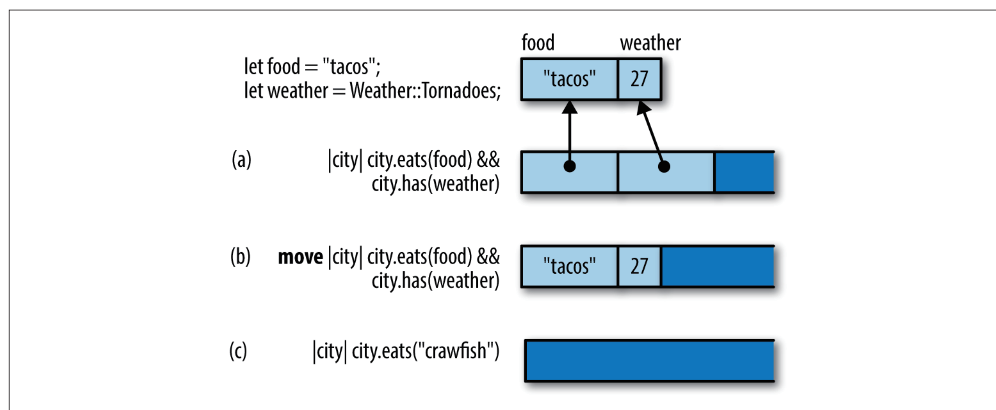


图 14-1：闭包的内存布局

闭包 (a) 使用了这两个变量。显然，这里是想查找既有玉米饼（taco）又有龙卷风（tornado）的城市。在内存中，这个闭包类似于一个小结构体，其包含对它所使用变量的引用。

注意，这里没有包含指向其代码的指针。这是不必要的，只要 Rust 知道闭包的类型，就知

道调用它时该执行什么代码。

闭包 (b) 几乎完全相同，只不过它是一个转移闭包，因此会包含实际的值而非引用。

闭包 (c) 没有用到其环境中的任何变量。此时结构体是空的，因此这个闭包根本不会占用内存。

如图所示，这些闭包都不怎么占空间。但就算那么点字节在实践中也不是总会用到。编译器经常会把对闭包的调用行内化，这样就连图中所示的小结构体也会被优化掉。

14.5 节将介绍如何在堆上分配闭包并通过特型对象动态调用它们。这样会稍微慢一点，但仍然跟其他特型对象一样快。

14.4　闭包和安全

接下来主要介绍闭包与 Rust 安全系统的关系。正如本章前面所讲的，闭包主要就是在创建的时候可能转移或借用被捕获的变量。但这样造成的影响并不十分明显，特别是在闭包清除或修改捕获的值时。

14.4.1　杀值的闭包

前面介绍了借用值的闭包，也介绍了盗用值的闭包，接下来将介绍杀值的闭包。

当然，“杀值”并不是真正恰当的术语。在 Rust 中，我们说**清除**（drop）值。最直观的方式就是调用 `drop()`：

```
let my_str = "hello".to_string();
let f = || drop(my_str);
```

在调用 `f` 时，`my_str` 会被清除。

如果调用它两次会怎么样？

```
f();
f();
```

下面来分析一下。第一次调用 `f`，它清除了 `my_str`，这意味着会释放存储字符串的内存，交还给系统。第二次调用 `f`，同样的操作又会执行一遍。这就是 C++ 中会触发未定义行为的经典错误：**重复释放**（double free）。

在 Rust 中清除一个 `String` 两次也是件坏事。好在 Rust 并不容易欺骗：

```
f();  // 没问题
f();  // 错误：使用转移的值
```

Rust 知道这个闭包不能被调用两次。

一个闭包只能被调用一次，这看起来确实是一件离奇的事。但本书已经把所有权和生命期介绍得非常详细了。值会被用尽（即转移）是 Rust 的一个核心概念。这个概念适用于 Rust 中的一切，闭包当然也不例外。

14.4.2 FnOnce

下面再试试欺骗 Rust，让它两次清除一个 String。这次使用的泛型函数如下：

```
fn call_twice<F>(closure: F) where F: Fn() {
    closure();
    closure();
}
```

可以给这个函数传入任何实现 Fn() 特型的闭包。换句话说，就是不接收参数且返回 () 的闭包。（与函数一样，如果返回值类型是 () 则可以省略。Fn() 是对 Fn() -> () 的简写。）

接下来，如果把前面那个不安全的闭包传给这个泛型函数会怎样？

```
let my_str = "hello".to_string();
let f = || drop(my_str);
call_twice(f);
```

同样，这个闭包在被调用时会清除 my_str。调用它两次就是重复释放。但这次，Rust 又没有上当：

```
error[E0525]: expected a closure that implements the `Fn` trait, but
              this closure only implements `FnOnce`
  --> closures_twice.rs:12:13
   |
12 |     let f = || drop(my_str);
   |             ^^^^^^^^^^^^^^^
   |
note: the requirement to implement `Fn` derives from here
  --> closures_twice.rs:13:5
   |
13 |     call_twice(f);
   |     ^^^^^^^^^^
```

这里的错误消息已经告诉了我们 Rust 是如何处理"杀值闭包"的。这种闭包从语言层面上已经被完全禁止了，但负责清理工作的闭包有时候也是有用的。为此 Rust 会限制这种闭包的使用。像 f 这样清除值的闭包不能是 Fn。更确切地说，它们根本就不具备 Fn 的资格。它们实现的是没有那么通用的特型 FnOnce，这种特型的闭包只能调用一次。

第一次调用 FnOnce 闭包时，**闭包本身也会被用掉**。就像这两个特型 Fn 和 FnOnce 是这样定义的一样：

```
// 特型Fn和FnOnce的伪代码实现，没有参数
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

跟 a + b 这样的算术表达式实际上是对方法调用 Add::add(a, b) 的简写一样，Rust 将 closure() 作为对前面展示的两个特型方法之一的简写。对于 Fn 闭包，closure() 会扩展

为 closure().call()，这个方法以自身的引用作为参数，因此这个闭包不会被转移。但如果闭包只能安全地调用一次，即对于 FnOnce 闭包，closure() 则会扩展为 closure().call_once()。这个方法会取得 self 的值，所以闭包会被用掉。

当然，这里是有意使用 drop() 来制造问题。实践中，多数时候只会偶尔碰到前面的问题。虽然不常出现，但有时候也会写出意外把某个值用掉的闭包来：

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict { // 哎呀!
        println!("{:?} - {:?}", key, value);
    }
};
```

然后，如果调用 debug_dump_dict() 超过一次，就会看到类似下面这样的错误消息：

```
error[E0382]: use of moved value: `debug_dump_dict`
  --> closures_debug_dump_dict.rs:18:5
   |
17 |     debug_dump_dict();
   |     --------------- value moved here
18 |     debug_dump_dict();
   |     ^^^^^^^^^^^^^^^ value used here after move
   |
   = help: closure was moved because it only implements `FnOnce`
```

为找到问题所在，必须知道这个闭包为什么是 FnOnce。这里把哪个值用掉了？闭包里唯一引用的不就是 dict 吗？啊，发现问题了：这里直接迭代 dict 导致把它给用掉了。应该使用 &dict 而不是 dict，即要访问值的引用：

```
let debug_dump_dict = || {
    for (key, value) in &dict { // 不会用掉dict
        println!("{:?} - {:?}", key, value);
    }
};
```

这样就修复了问题。现在这个函数变成了 Fn，调用多少次都不会出问题了。

14.4.3 FnMut

还有一种闭包，就是包含可修改数据或 mut 引用的闭包。

Rust 认为非 mut 值可以安全地在线程间共享。但是，如果共享的非 mut 闭包里包含 mut 数据同样是不安全的。在多个线程里调用这个闭包会导致各种资源争用问题，与多个线程同时读写相同的数据一样。

因此，Rust 又定义了一种名为 FnMut 的闭包，即可以写数据的闭包。FnMut 闭包要使用 mut 引用来调用，其定义类似如下这样：

```
// 特型Fn、FnMut和FnOnce的伪代码实现
trait Fn() -> R {
    fn call(&self) -> R;
```

```
}

trait FnMut() -> R {
    fn call_mut(&mut self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

任何需要以 mut 方式访问值但不会清除任何值的闭包都是 FnMut 闭包。例如：

```
let mut i = 0;
let incr = || {
    i += 1; // 这里会借用对i的可修改引用
    println!("Ding! i is now: {}", i);
};
call_twice(incr);
```

前面定义的 call_twice 需要一个 Fn。因为 incr 是 FnMut 而不是 Fn，所以上面的代码会编译失败。不过要修复也很简单。为理解修复的原理，先后退一步，回顾一下 Rust 的这 3 种闭包。

- Fn 是没有调用次数限制的闭包和函数，是所有 fn 函数中最高的一种。
- FnMut 是如果闭包本身声明为 mut 也可以多次调用的闭包。
- FnOnce 是如果调用者拥有闭包则只能调用一次的闭包。

每个 Fn 都满足 FnMut 的要求，每个 FnMut 都满足 FnOnce 的要求。如图 14-2 所示，它们不是相互独立的。

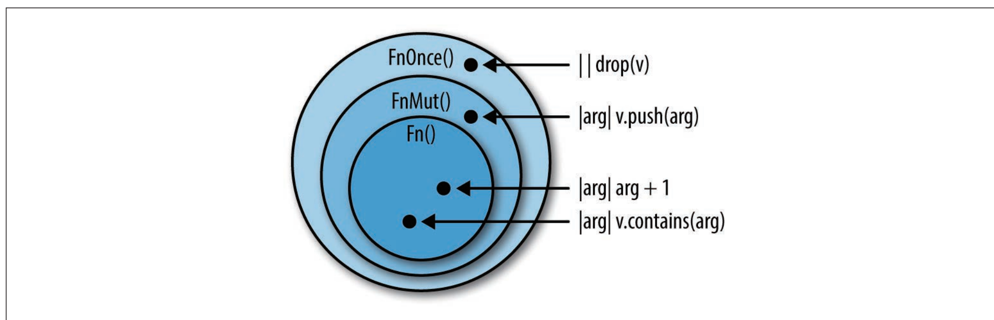


图 14-2：3 种闭包的维恩图

实际上，Fn() 是 FnMut() 的子特型，而 FnMut() 又是 FnOnce() 的子特型。于是 Fn 就成了最专一且最强大的类别。FnMut 和 FnOnce 则是包含使用限制的更广泛的类别。

现在，我们已经了解了这 3 种闭包。很显然，要接收最为广泛的闭包，call_twice 函数应该接收所有 FnMut 闭包，也就是要修改为这样：

```
fn call_twice<F>(mut closure: F) where F: FnMut() {
    closure();
```

```
        closure();
    }
```

第一行中的绑定之前是 F: Fn()，而现在是 F: FnMut()。这样一改，仍然可以接收所有 Fn 闭包，而且还能对修改数据的闭包调用 call_twice：

```
let mut i = 0;
call_twice(|| i += 1); // 没问题!
assert_eq!(i, 2);
```

14.5 回调

很多库提供的 API 会使用**回调**，也就是用户提供的函数，供库在以后调用。事实上，本书示例中也出现过这种 API。第 2 章曾使用 Icon 框架编写过一个简单的 Web 服务器，代码类似这样：

```
fn main() {
    let mut router = Router::new();

    router.get("/", get_form, "root");
    router.post("/gcd", post_gcd, "gcd");

    println!("Serving on http://localhost:3000...");
    Iron::new(router).http("localhost:3000").unwrap();
}
```

这个路由器的用途是将来自互联网的请求转发给处理特定请求的 Rust 代码。在这个例子中，get_form 和 post_gcd 就是程序其他地方用 fn 关键字声明的两个函数名。但其实也可以在这里传入闭包，比如：

```
let mut router = Router::new();

router.get("/", |_: &mut Request| {
    Ok(get_form_response())
}, "root");
router.post("/gcd", |request: &mut Request| {
    let numbers = get_numbers(request)?;
    Ok(get_gcd_response(numbers))
}, "gcd");
```

这是因为 Iron 的代码可以接收任何线程安全的 Fn 作为参数。

如果是自己的程序，应该怎么实现呢？可以从头开始写一个非常简单的路由器，不使用任何 Iron 的代码。先声明几种类型表示 HTTP 的请求和响应：

```
struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}
```

```
struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}
```

当前路由器的任务就是简单地保存一个包含 URL 到回调的映射的表，以便可以按需调用正确的回调。（简单起见，只允许用户创建与一个 URL 匹配的路由。）

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// 创建一个空路由器
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }

    /// 给路由器添加一个路由
    fn add_route(&mut self, url: &str, callback: C) {
        self.routes.insert(url.to_string(), callback);
    }
}
```

可是，有一个地方有错误。你发现了吗？

如果只给这个路由器添加一个路由是没有问题的：

```
let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());
```

这可以通过编译并运行。不过，如果再添加一个路由：

```
router.add_route("/gcd", |req| get_gcd_response(req));
```

就会看到错误：

```
error[E0308]: mismatched types
  --> closures_bad_router.rs:41:30
   |
41 |     router.add_route("/gcd", |req| get_gcd_response(req));
   |                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |                              expected closure, found a different closure
   |
   = note: expected type `[closure@closures_bad_router.rs:40:27: 40:50]`
              found type `[closure@closures_bad_router.rs:41:30: 41:57]`
note: no two closures, even if identical, have the same type
help: consider boxing your closure and/or using it as a trait object
```

我们说的错误就在定义 `BasicRouter` 类型上：

```
struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}
```

这里无意之间把 BasicRouter 声明为了只有一个回调类型 C，HashMap 中的所有回调都是这个类型。11.1.4 节展示过一个有同样问题的 Salad 类型：

```
struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

解决方案与 Salad 类型一样：因为想要支持多种类型，所以这里需要装箱和特型目标：

```
type BoxedCallback = Box<Fn(&Request) -> Response>;

struct BasicRouter {
    routes: HashMap<String, BoxedCallback>
}
```

每个“箱子”可以包含不同类型的闭包，因此一个 HashMap 可以包含所有类型的回调。注意类型参数 C 不见了。

这需要对方法也做一些调整：

```
impl BasicRouter {
    // 创建一个空路由器
    fn new() -> BasicRouter {
        BasicRouter { routes: HashMap::new() }
    }

    /// 给路由器添加一个路由
    fn add_route<C>(&mut self, url: &str, callback: C)
        where C: Fn(&Request) -> Response + 'static
    {
        self.routes.insert(url.to_string(), Box::new(callback));
    }
}
```

（注意 add_route 中类型签名 C 的两个绑定：Fn 特型和 'static 生命期。Rust 支持添加这个 'static 绑定。没有它，调用 Box::new(callback) 就会出错，因为如果闭包中包含会超出作用域的变量的引用，那么这个调用就是不安全的。）

最后，我们简单的路由器就可以处理请求了：

```
impl BasicRouter {
    fn handle_request(&self, request: &Request) -> Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request)
        }
    }
}
```

14.6　有效使用闭包

如前所见，Rust 的闭包与大多数其他语言中的闭包不同。最大的区别是在有垃圾回收的语言中，可以在闭包中使用局部变量而无须考虑生命期或所有权。如果没有垃圾回收，情况

就不同了。Java、C# 和 JavaScript 中常见的设计模式无法原封不动照搬到 Rust 里来。

以图 14-3 所示的 MVC（Model-View-Controller，模型 – 视图 – 控制器）设计模式为例。对用户界面上的每个元素，MVC 框架都会创建 3 个对象：模型、视图和控制器，其中**模型**表示 UI 元素的状态，**视图**负责元素的外观，而**控制器**处理用户交互。多年来，MVC 已经出现了数不清的变体。但核心还是 3 个对象共同分担 UI 的职责。

那么问题来了。通常，每个对象都会有另一个或另两个对象的引用。可能是直接引用，也可能是通过回调来引用，如图 14-3 所示。在 3 个对象中的一个对象发生了某个事件时，它会通知另外两个对象，因此一切会立即更新。问题在于，哪个对象“拥有”其他对象则永远说不清。

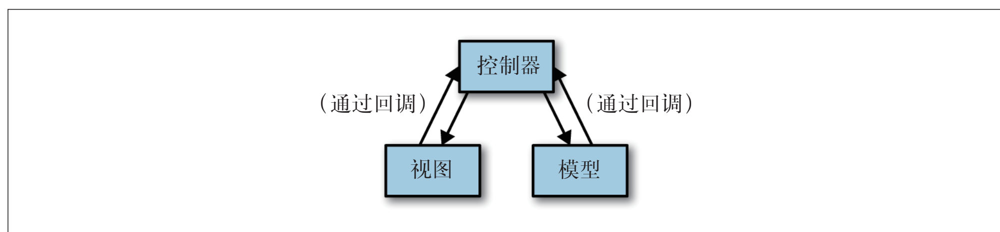


图 14-3：模型 – 视图 – 控制器设计模式

这个模式在未经修改的情况下无法在 Rust 中直接使用。在 Rust 中，必须明确所有权，必须消除循环引用。模型和控制器不能直接相互引用。

Rust 激进的赌注就是一定存在优秀的替代设计。有时候，可以让每个闭包接收它需要的引用作为参数，通过闭包所有权和生命期来解决问题。有时候，可以在系统中给每件东西分配一个数值，然后传递数值而不传递引用。或者，可以实现诸多 MVC 变体中的一种，保证对象之间并不是都相互引用。再或者，可以仿效某个非 MVC 系统，比如 Facebook 的 Flux 架构，实现单向数据流，如图 14-4 所示。

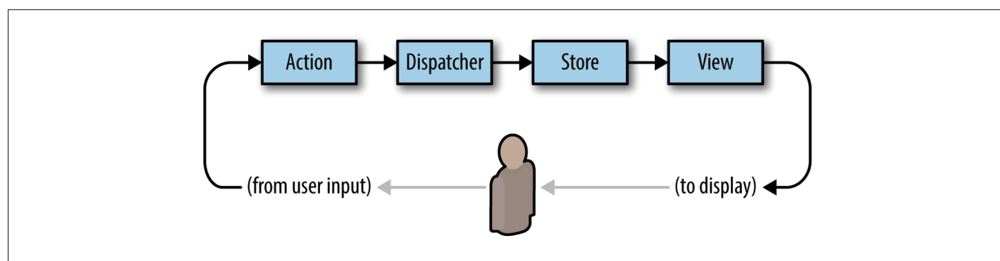


图 14-4：Flux 架构是 MVC 的一种替代架构

简言之，如果想使用 Rust 闭包制造“对象之海”，就会碰到各种问题。但是任何问题都不止一种解决方案。从这个意义上说，软件工程这门学科已经转向了替代方案，因为它们更简单。

下一章会讨论闭包能够真正大显身手的主题。届时，我们会利用 Rust 闭包的简洁、速度和高效写出一种不同风格的代码，这种代码写起来好玩，看起来好懂，而且极为实用。

第15章

迭代器

一直重复的一天终于结束了。

——Phil（电影《土拨鼠节》，又译《偷天情缘》）

迭代器是一个可以产生一系列值的值，通常要使用循环来操作。Rust 标准库提供了遍历向量、字符串、散列表和其他集合的迭代器，也提供了从输入流、网络连接和线程间通信渠道产生文本行的迭代器。当然，你也可以实现自己的迭代器。Rust 的 `for` 循环是使用迭代器最自然的语法，但迭代器本身也提供了丰富的映射、过滤、拼接、收集等方法。

Rust 的迭代器非常灵活、擅长表达且富有效率。来看下面的函数，它返回前 *n* 位正整数的和（通常叫第 *n* 个**三角形数**）：

```
fn triangle(n: i32) -> i32 {
    let mut sum = 0;
    for i in 1..n+1 {
        sum += i;
    }
    sum
}
```

表达式 `1..n+1` 是一个 `Range<i32>` 值。`Range<i32>` 是一个迭代器，可以产生从起始值（包含）开始到终止值（不含）结束的整数序列。因此，可以在 `for` 循环中使用它作为操作数来计算从 1 到 *n* 的和。

不过迭代器也有一个 `fold` 方法，该方法可以用来实现同样的逻辑：

```
fn triangle(n: i32) -> i32 {
    (1..n+1).fold(0, |sum, item| sum + item)
}
```

这样累加值会从 0 开始，而 `fold` 取得 `1..n+1` 产生的每个值，把累加值和这个值传给闭包

`|sum, item| sum + item`。然后闭包的返回值又作为新的累加值。最后返回的值就是 `fold` 自己返回的值，在这里就是整个序列的总和。习惯了使用 `for` 和 `while` 循环可能会觉得这样有些奇怪，不过熟悉之后就会发现 `fold` 既清晰又简洁。

这是标准的函数式编程风格，具有表达性的优势。不过 Rust 的迭代器经过了认真设计，可以确保编译器也把它们编译为优化的机器码。对于前面第二版定义的发布构建，Rust 知道 `fold` 的定义，会将其直接嵌入 `triangle` 中。此外，闭包 `|sum, item| sum + item` 也会嵌入其中。最后，Rust 检查了组合的代码，发现有计算从 1 加到 *n* 的更简单方式：和始终等于 `n * (n+1) / 2`。Rust 会把 `triangle` 的整个函数体、循环、闭包以及所有一切都转换为一条乘法指令和几个算术操作。

这个例子恰好包含简单算术，实际上对于更复杂的计算迭代器同样有优异表现。这也是 Rust 提供灵活抽象，但没有或几乎没有开销的典型例证。

本章剩下的内容分为 5 部分。

- 首先会解释 `Iterator` 和 `IntoIterator` 特型，它们是 Rust 迭代器的基础。
- 然后会看看典型迭代器流水线的 3 个阶段：基于某些来源的值创建迭代器、通过选择和处理值来产生新迭代器，以及消费迭代器产生的值。
- 最后会展示如何实现自己的迭代器。

迭代器的方法非常多，因此如果你领会了中心思想，大可跳过某一节。不过，迭代器是写出 Rust 特色代码的常用特性，熟悉迭代器提供的这些工具对掌握这门语言是必不可少的。

15.1 Iterator和IntoIterator特型

迭代器指的是任何实现 `std::iter::Iterator` 特型的值。

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ... // 许多默认方法
}
```

`Item` 是迭代器产生值的类型。`next` 方法要么返回 `Some(v)`，其中 `v` 是迭代器的下一个值；要么返回 `None`，表示序列终止。这里省略了迭代器的许多默认方法，本章后面会详细介绍。

如果有一种自然的方式可以迭代某种类型，那它可以实现 `std::iter::IntoIterator`。它的 `into_iter` 方法接收一个值，然后基于这个值返回一个迭代器：

```
trait IntoIterator where Self::IntoIter::Item == Self::Item {
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

`IntoIter` 是迭代器值本身的类型，而 `Item` 是它产生值的类型。我们称任何实现 `IntoIterator` 的类型为**可迭代类型**（iterable），因为可以在需要的时候通过循环来访问它。

Rust 的 `for` 循环可以顺畅地实现迭代器操作。要迭代某个向量的元素，可以这样写：

```
println!("There's:");
let v = vec!["antimony", "arsenic", "aluminum", "selenium"];

for element in &v {
    println!("{}", element);
}
```

而在后台，每个 `for` 循环都只是对调用 `IntoIterator` 和 `Iterator` 方法的简写形式：

```
let mut iterator = (&v).into_iter();
while let Some(element) = iterator.next() {
    println!("{}", element);
}
```

这也就是说，`for` 循环会使用 `IntoIterator::into_iter` 将其操作数 `&v` 转换为一个迭代器，然后重复调用 `Iterator::next`。每次返回 `Some(element)`，`for` 循环都执行其循环体。而如果返回的是 `None`，则结束循环。

虽然 `for` 循环始终会对其操作数调用 `into_iter`，但也可以直接把迭代器传给 `for` 循环。前面例子中循环访问 `Range` 时就是这样做的。所有迭代器都自动实现 `IntoIterator`，因而有一个 `into_iter` 方法返回这个迭代器。

如果在迭代器返回 `None` 之后再次调用它的 `next` 方法，`Iterator` 特型则并没有规定这时候该怎么做。大多数迭代器会再次返回 `None`，但不是全部。（如果你因此碰到了问题，15.3.7 节的 `fuse` 适配器可能会有用。）

下面简单总结一下迭代器相关的术语。

- 如前所述，**迭代器**（iterator）指的是任何实现 `Iterator` 的类型。
- **可迭代类型**（iterable）指的是实现 `IntoIterator` 的类型：调用它的 `into_iter` 方法可以取得它的迭代器。因此前面向量的引用 `&v` 就是一个可迭代类型。
- 迭代器**产生**值。
- 迭代器产生的值叫**迭代项**(item)。在前面的例子中，`"antimony"`、`"arsenic"` 等都是迭代项。
- 接收迭代器产生的迭代项的代码叫**消费者**（consumer）。前面例子中的 `for` 循环消费了迭代器的迭代项。

15.2 创建迭代器

Rust 标准库文档详细解释了每个类型提供了什么样的迭代器，但文档遵循某种通用形式以便读者理清头绪，找到想找的内容。

15.2.1 iter和iter_mut方法

大多数集合类型提供了 `iter` 和 `iter_mut` 方法，返回该类型的迭代器，产生每个迭代项的共享或可修改引用。切片类型 `&[T]` 和 `&str` 也有 `iter` 和 `iter_mut` 方法。如果不想使用 `for` 循环，那么这两个方法就是取得迭代器最常用的方式：

```
let v = vec![4, 20, 12, 8, 6];
let mut iterator = v.iter();
```

```
assert_eq!(iterator.next(), Some(&4));
assert_eq!(iterator.next(), Some(&20));
assert_eq!(iterator.next(), Some(&12));
assert_eq!(iterator.next(), Some(&8));
assert_eq!(iterator.next(), Some(&6));
assert_eq!(iterator.next(), None);
```

这个迭代器的迭代项类型是 `&i32`，每次调用 `next` 都会产生对下一个元素的引用，直到到达向量的末尾。

每个类型都可以实现 `iter` 和 `iter_mut`，什么实现方式合适自己决定。`std::path::Path` 的 `iter` 方法返回的迭代器每次会产生路径的一个组件：

```
use std::ffi::OsStr;
use std::path::Path;

let path = Path::new("C:/Users/JimB/Downloads/Fedora.iso");
let mut iterator = path.iter();
assert_eq!(iterator.next(), Some(OsStr::new("C:")));
assert_eq!(iterator.next(), Some(OsStr::new("Users")));
assert_eq!(iterator.next(), Some(OsStr::new("JimB")));
...
```

这个迭代器的迭代项类型是 `&std::ffi::OsStr`，是操作系统调用可以接受的、借用的字符串切片。

15.2.2　IntoIterator实现

如果类型实现了 `IntoIterator`，则可以调用它的 `into_iter` 方法，跟 `for` 循环一样：

```
// 一般会使用HashSet，但它的迭代顺序不确定。为保证顺序，
// 因此这个例子使用了BTreeSet
use std::collections::BTreeSet;
let mut favorites = BTreeSet::new();
favorites.insert("Lucy in the Sky With Diamonds".to_string());
favorites.insert("Liebesträume No. 3".to_string());

let mut it = favorites.into_iter();
assert_eq!(it.next(), Some("Liebesträume No. 3".to_string()));
assert_eq!(it.next(), Some("Lucy in the Sky With Diamonds".to_string()));
assert_eq!(it.next(), None);
```

实际上大多数集合提供了不止一个 `IntoIterator` 的实现，分别用于共享引用、可修改引用和转移。

- 对于集合的**共享引用**，`into_iter` 会返回产生迭代项共享引用的迭代器。例如，在前面的代码中，`(&favorites).into_iter()` 会返回一个 `Item` 类型是 `&String` 的迭代器。
- 对于集合的**可修改引用**，`into_iter` 会返回产生迭代项可修改引用的迭代器。例如，如果 `vector` 是某种 `Vec<String>`，那么调用 `(&mut vector).into_iter()` 会返回一个 `Item` 类型是 `&mut String` 的迭代器。
- 对于**按值**传递的集合，`into_iter` 返回的迭代器会取得集合的所有权，并按值返回迭代项。此时，迭代项的所有权从集合转移到消费者，原始集合在这个过程中会被消费掉。例如，

前面例子中调用 favorites.into_iter() 返回的迭代器会按值产生每个字符串，消费者会取得每个字符串的值。当迭代器被清除后，BTreeSet 中剩余的元素也会被清除，而集合所剩的空壳也会被处理掉。

因为 for 循环会对其操作数应用 IntoIterator::into_iter，所以这 3 个实现就可以支持迭代集合的共享引用或可修改引用，以及消费集合并取得其元素所有权：

```
for element in &collection { ... }
for element in &mut collection { ... }
for element in collection { ... }
```

每种用法都会对应调用上面的一种 IntoIterator 实现。

并不是所有类型都会提供全部 3 种实现。例如，HashSet、BTreeSet 和 BinaryHeap 就没有对可修改引用实现 IntoIterator，因为修改它们的元素可能违背类型的不变性，比如被修改的值变成不同的散列值或者相对周边元素做了不同排序，导致修改后把元素放到错误的位置上。其他类型确实有支持可修改引用的，但只是部分支持。例如，HashMap 和 BTreeMap 会产生它们值的可修改引用，但只产生它们键的共享引用，原因与前面所讲类似。

一般原则是迭代应该高效和可预测，因此相比于提供耗时或可能导致意外行为（例如，重新计算 HashSet 元素的散列之后又会重新访问它们）的实现，Rust 选择完全忽略它们。

切片实现了 3 种 IntoIterator 变体中的两个，因为它本身没有元素的所有权，所以就没有“按值”这种情况。而 &[T] 和 &mut [T] 的 into_iter 返回的迭代器，可以产生对它们元素的共享引用和可修改引用。如果把底层切片类型 [T] 想象为某种类型的集合，就能从整体上理解这种实现了。

有人可能注意到了，对于前两个 IntoIterator 变体，即共享引用和可修改引用，等价于在引用值上调用 iter 或 iter_mut。Rust 为什么两个都提供呢？

IntoIterator 是 for 循环底层的基础，因此显然是必要的。但在不使用 for 循环时，favorites.iter() 要比 (&favorites).into_iter() 清晰得多。按共享引用迭代是很常见的，所以 iter 和 iter_mut 更符合人的常识。

IntoIterator 在泛型代码中也可以派上用场。比如，可以通过 T: IntoIterator 绑定来限制类型变量 T 为可迭代的类型。或者，可以通过 T: IntoIterator<Item=U> 进一步要求迭代产生指定的类型 U。例如，下面这个函数可以接收任何可迭代类型，将它们通过 "{:?}" 格式打印的值打印出来：

```
use std::fmt::Debug;

fn dump<T, U>(t: T)
    where T: IntoIterator<Item=U>,
          U: Debug
{
    for u in t {
        println!("{:?}", u);
    }
}
```

不能使用 iter 和 iter_mut 来实现这个泛型函数，因为它们不是任何特型的方法。大多数可迭代类型只是恰好有以这两个名字命名的方法而已。

15.2.3　drain方法

很多集合类型会提供 drain 方法，以一个集合的可修改引用为参数，返回一个能把每个元素所有权传给消费者的迭代器。不过，与按值接收并消费集合的 into_iter() 方法不同，drain 只是借用对集合的引用，在迭代器被清除后，它会清空集合中所有剩余的元素。

在可以通过范围指定索引的类型如 String、向量和 VecDeque 中，drain 方法接收要移除元素的范围，而不是排取（drain）整个序列：

```
use std::iter::FromIterator;

let mut outer = "Earth".to_string();
let inner = String::from_iter(outer.drain(1..4));

assert_eq!(outer, "Eh");
assert_eq!(inner, "art");
```

如果确实需要排取整个序列，那么可以使用全范围（..）作为参数。

15.2.4　其他迭代器源

上一节主要介绍了向量和 HashMap 这样的集合类型。实际上，标准库中还有很多其他类型支持迭代。表 15-1 总结了其中比较有意思的，但并不止这些。第 16 章、第 17 章和第 18 章在介绍特定类型时会详细介绍其他方法。

表15-1：标准库中的其他迭代器

类型或特型	表　达　式	说　　明
std::ops::Range	1..10	端点必须是可以迭代的整数类型。范围包含起始值，不包含终止值
std::ops::RangeFrom	1..	无限定迭代。起始值必须是整数。如果值超过类型限制可能会诧异或溢出
Option<T>	Some(10).iter()	类似长度为 0（None）或 1（Some(v)）的向量
Result<T, E>	Ok("blah").iter()	类似 Option，产生 Ok 值
Vec<T>, &[T]	v.windows(16)	从左到右产生给定长度的每个连续切片。窗口重叠
	v.chunks(16)	从左到右产生给定长度的非重叠连续切片
	v.chunks_mut(1024)	类似 chunks，但切片是可修改的
	v.split(|byte| byte & 1 != 0)	产生以匹配给定断言的元素分割的切片
	v.split_mut(...)	同上，但产生可修改切片
	v.rsplit(...)	类似 split，但从右到左产生切片
	v.splitn(n, ...)	类似 split，但最多产生 *n* 个切片

（续）

类型或特型	表　达　式	说　　明
`String, &str`	`s.bytes()`	产生 UTF-8 形式的字节
	`s.chars()`	产生 UTF-8 表示的字符
	`s.split_whitespace()`	以空格分割字符串，产生非空格字符的切片
	`s.lines()`	产生字符串行的切片
	`s.split('/')`	以给定模式分割字符串，产生匹配之间内容的切片。模式可以是字符、字符串、闭包等
	`s.matches(char::is_numeric)`	产生匹配给定模式的切片
`std::collections::HashMap,` `std::collections::BTreeMap`	`map.keys(),` `map.values()`	产生对映射键或值的共享引用
	`map.values_mut()`	产生对条目值的可修改引用
`std::collections::HashSet,` `std::collections::BTreeSet`	`set1.union(set2)`	产生对 `set1` 与 `set2` 合集元素的共享引用
	`set1.intersection(set2)`	产生对 `set1` 与 `set2` 交集元素的共享引用
`std::sync::mpsc::Receiver`	`recv.iter()`	产生另一个线程中对应 `Sender` 发送的值
`std::io::Read`	`stream.bytes()`	从 I/O 流产生字节
	`stream.chars()`	将流作为 UTF-8 解析并产生字符
`std::io::BufRead`	`bufstream.lines()`	将流作为 UTF-8 解析并产生 `String` 行
	`bufstream.split(0)`	以给定字节分割流，产生该字节间的 `Vec<u8>` 缓冲
`std::fs::ReadDir`	`std::fs::read_dir(path)`	产生目录项
`std::net::TcpListener`	`listener.incoming()`	产生到来的网络连接
`Free functions`	`std::iter::empty()`	立即返回 `None`
	`std::iter::once(5)`	产生给定的值，然后结束
	`std::iter::repeat("#9")`	一直产生给定的值

15.3　迭代器适配器

一旦有了迭代器，`Iterator` 特型就会提供大量可供选择的**适配器方法**，或简称**适配器**（adapter）。这些适配器消费一个迭代器并创建一个具备有用行为的新迭代器。为了解适配器，接下来会介绍其中两个最流行的。

15.3.1　map和filter

`Iterator` 特型的 `map` 适配器可以为迭代器的每个迭代项都应用一个闭包，而 `filter` 适配器可以通过迭代器来过滤某些迭代项，使用闭包来决定保留哪个、清除哪个。

例如，假设我们在迭代文本行时想忽略每一行开头和末尾的空格。标准库的 `str::trim` 方法会清除 `&str` 开头和末尾的空格，返回一个新的修整后的借用原始值 `&str`。可以通过 `map` 适配器给迭代器中的每一行应用 `str::trim`：

```
let text = "  ponies  \n   giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .collect();
assert_eq!(v, ["ponies", "giraffes", "iguanas", "squid"]);
```

text.lines() 调用返回的是产生字符串行的迭代器。在这个迭代器上调用 map 会返回另一个迭代器，它的每一项是对每一行应用 str::trim 之后的结果。最后，collect 将所有项收集到一个向量里。

当然，调用 map 返回的迭代器本身仍然可以继续使用适配器。对于前面的例子，如果还想在结果中排除美洲蜥蜴（"iguanas"），可以这样写：

```
let text = "  ponies  \n   giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .filter(|s| *s != "iguanas")
    .collect();
assert_eq!(v, ["ponies", "giraffes", "squid"]);
```

这里，调用 filter 会返回第三个迭代器，只产生 map 迭代器结果中对闭包 |s| *s != "iguanas" 返回 true 的那些项。这样一个迭代器适配器的连缀调用，就像 Unix 中的管道操作。每个适配器都有自己独立的职责，从左到右可以清楚地知道每次转换都做了什么。

map 和 filter 适配器的签名如下：

```
fn map<B, F>(self, f: F) -> some Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> B;

fn filter<P>(self, predicate: P) -> some Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

这里用作返回值类型的 some Iterator<...> 并不是有效的 Rust 语法[1]。真正的返回值类型是不透明的 struct 类型，但表达性差；关键是实践中这些方法会返回给定 Item 类型的迭代器。

因为大多数适配器按值取得 self，所以它们要求 Self 是 Sized（所有常用迭代器都是）。

map 迭代器将每一项按值传给其闭包，进而将闭包返回结果的所有权传给消费者。filter 迭代器将每一项的共享引用传给其闭包，在将选中项传给其消费者时，保留该项的所有权。这也正是代码中会解引用 s 再将其与 "iguanas" 比较的原因。filter 迭代器的项类型是 &str，因此闭包参数 s 的类型是 &&str。

关于迭代器适配器，有两个要点需要明确。

首先，简单地在一个迭代器上调用适配器不会消费任何项，只会返回一个新迭代器，并可以按需从第一个迭代器排取以产生自己的项。在适配器链中，唯一可以真正让操作落地的是在最终的迭代器上调用 next。

注 1：Rust RFC 1522 会添加与这里的 some Iterator 表示法非常类似的语法。Rust 1.17 还没有默认包含这个语法。

因此在前面的例子中，方法调用 `text.lines()` 本身不会真正从字符串中解析出任何行，而只会返回一个迭代器，**将来**再根据需要解析行。类似地，`map` 和 `filter` 也会返回新迭代器，**将来**再根据需要映射和过滤。在 `collect` 调用 `filter` 迭代器的 `next` 之前，不会发生真正的操作。

理解这一点对使用具有副作用的适配器特别重要。例如，下面的代码什么也不会打印：

```
["earth", "water", "air", "fire"]
    .iter().map(|elt| println!("{}", elt));
```

`iter` 调用返回基于数组元素的迭代器，而 `map` 调用返回对第一个迭代器产生的每个值应用闭包的第二个迭代器。但调用链中并没有真正请求处理某个值的代码，因此也不会调用任何迭代器的 `next` 方法。事实上，Rust 对此会给出警告：

```
warning: unused result which must be used:
iterator adaptors are lazy and do nothing unless consumed
    |
387 | /         ["earth", "water", "air", "fire"]
388 | |             .iter().map(|elt| println!("{}", elt));
    | |___________________________________________________^
    |
= note: #[warn(unused_must_use)] on by default
```

错误消息中的“lazy”（懒）在这里并不是一个贬义词，而是一个专用概念，指的是一种将计算推迟到真正需要时的机制。Rust 的约定是迭代器在满足每个 `next` 调用前应该尽量什么都不做。在这个例子中，根本就没有调用 `next`，所以什么也不会发生。

第二个要点是迭代器适配器属于零开销抽象。因为 `map`、`filter` 及其同类方法是泛型的，所以把它们应用给迭代器会针对涉及的特定迭代器类型特化它们的代码。这意味着 Rust 有足够的信息把每个迭代器的 `next` 方法行内化到其消费者中，然后将整套代码作为一个单位翻译为机器码。因此前面所示迭代器的 `lines/map/filter` 调用链，实际上与手写的如下代码同样高效：

```
for line in text.lines() {
    let line = line.trim();
    if line != "iguanas" {
        v.push(line);
    }
}
```

本节剩下的部分介绍 `Iterator` 特型支持的各种适配器。

15.3.2 filter_map和flat_map

`map` 适配器适合一个进项对应一个出项的场景。有时候，可能需要在迭代中清除某些项，或者用零或多项替代一项。此时 `filter_map` 和 `flat_map` 适配器可以满足需求。

`filter_map` 适配器与 `map` 类似，只是它允许闭包在迭代过程中要么转换项（像 `map` 那样），要么删除项。总之就类似 `filter` 和 `map` 的组合。它的签名如下：

```
fn filter_map<B, F>(self, f: F) -> some Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> Option<B>;
```

跟 `map` 的签名一样，只是这里闭包返回的是 `Option<B>`，而不是 `B`。如果闭包返回 `None`，就表示从迭代中清除项；如果返回 `Some(b)`，则 `b` 就是 `filter_map` 迭代器产生的下一项。

例如我们想处理一个空格分隔的字符串，从中解析出数值，然后处理数值，清除其他非数值。可以这样写：

```
use std::str::FromStr;

let text = "1\nfrond .25  289\n3.1415 estuary\n";
for number in text.split_whitespace()
                  .filter_map(|w| f64::from_str(w).ok()) {
    println!("{:4.2}", number.sqrt());
}
```

这样会打印出如下结果：

```
1.00
0.50
17.00
1.77
```

提供给 `filter_map` 的闭包尝试使用 `f64::from_str` 解析每个空格分隔的切片，返回 `Result<f64, ParseFloatError>`，然后 `.ok()` 把它转换为 `Option<f64>`：解析错误变成了 `None`，而成功解析出的结果变成了 `Some(v)`。`filter_map` 迭代器清除所有 `None` 值，产生每个 `Some(v)` 的值 `v`。

把 `map` 和 `filter` 像这样融合为一个调用（而不是分别调用它们）的目的何在？ `filter_map` 适配器的价值在刚才那个例子里已经展示得很充分了。它表明：在迭代中决定是否包含某一项的最好方式，就是尝试实际去处理它。直接使用 `filter` 和 `map` 当然可以，就是显得有点笨了：结果变成了 `Some(v)`。`filter_map` 迭代器清除所有 `None` 值，产生每个 `Some(v)` 的值 `v`。

把 `map` 和 `filter` 像这样融合为一个调用（而不是分别调用它们）的目的何在？ `filter_map` 适配器的价值在刚才那个例子里已经展示得很充分了。它表明：在迭代中决定是否包含某一项的最好方式，就是尝试实际去处理它。直接使用 `filter` 和 `map` 当然可以，就是显得有点笨了：

```
text.split_whitespace()
    .map(|w| f64::from_str(w))
    .filter(|r| r.is_ok())
    .map(|r| r.unwrap())
```

可以把 `flat_map` 适配器看成 `map` 加 `filter_map`，只不过现在闭包不只是会返回一项（像 `map` 那样）或返回零或多项（像 `filter` 那样），而是会返回任意多个项的序列。`flat_map` 迭代器产生闭包返回序列的拼接结果。

`flat_map` 的签名如下所示：

```
fn flat_map<U, F>(self, f: F) -> some Iterator<Item=U::Item>
    where F: FnMut(Self::Item) -> U, U: IntoIterator;
```

传给 `flat_map` 的闭包必须返回一个可迭代类型，任何可迭代类型都可以。[2]

注 2：实际上，因为可以把 `Option` 看成包含一系列零或一项的可迭代类型，所以 `iterator.filter_map(closure)` 和 `iterator.flat_map(closure)` 在 `closure` 返回 `Option` 时是等价的。

例如，假设有一个表，其映射的是国家与它们的主要城市。给定一组国家，怎么遍历它们的主要城市？

```
use std::collections::HashMap;

let mut major_cities = HashMap::new();
major_cities.insert("Japan", vec!["Tokyo", "Kyoto"]);
major_cities.insert("The United States", vec!["Portland", "Nashville"]);
major_cities.insert("Brazil", vec!["São Paulo", "Brasília"]);
major_cities.insert("Kenya", vec!["Nairobi", "Mombasa"]);
major_cities.insert("The Netherlands", vec!["Amsterdam", "Utrecht"]);

let countries = ["Japan", "Brazil", "Kenya"];

for &city in countries.iter().flat_map(|country| &major_cities[country]) {
    println!("{}", city);
}
```

这样会打印出如下结果：

```
Tokyo
Kyoto
São Paulo
Brasília
Nairobi
Mombasa
```

对此，一种理解方式是对每个国家，先取得其城市的向量，然后把所有向量拼接为一个序列，再把它打印出来。

但别忘了迭代器很懒惰，只有 `for` 循环调用 `flat_map` 迭代器的 `next` 方法时才会实际发生操作。拼接完整序列的操作不会在内存中发生。相反，这里会有一个小状态机，从城市迭代器中取值，每次取一个，直到取完为止。此时，才会产生下一个国家的新城市迭代器。效果就类似一个嵌套循环，只是被装配成了迭代器来使用。

15.3.3 scan

`scan` 适配器类似于 `map`，区别在于它会传给闭包一个可修改的值，而且可以选择提前终止迭代。它接收一个初始状态值和一个闭包，闭包又接收一个对这个状态的可修改引用和底层迭代器的下一项。这个闭包必须返回 `Option`，`scan` 迭代器将其作为自己的下一项。

例如，下面这个迭代器链会对另一个迭代器的项求平方，并在和超过 10 时终止迭代。

```
let iter = (0..10)
    .scan(0, |sum, item| {
        *sum += item;
        if *sum > 10 {
            None
        } else {
            Some(item * item)
        }
    });
```

```
assert_eq!(iter.collect::<Vec<i32>>(), vec![0, 1, 4, 9, 16]);
```

闭包的 sum 参数是一个迭代器私有变量的可修改引用，在 scan 的第一个参数中初始化，这里就是 0。闭包会更新 *sum，检查它是否超出了限制，然后返回迭代器的下一个结果。

15.3.4 take和take_while

Iterator 特型的 take 和 take_while 适配器用于在取得一定项数之后或闭包决定中断时终止迭代。它们的签名如下所示：

```
fn take(self, n: usize) -> some Iterator<Item=Self::Item>
    where Self: Sized;

fn take_while<P>(self, predicate: P) -> some Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

这两个适配器都会取得一个迭代器的所有权，返回一个新迭代器，这个新迭代器会从第一项开始产生值，但可能提前终止序列。take 迭代器在产生最多 *n* 项后返回 None。take_while 迭代器对每一项应用 predicate，在遇到第一个 predicate 返回 false 的项时返回 None，后续每次调用 next 也都返回 None。

例如，有一封邮件，其以空行分隔标题行和正文。可以使用 take_while 只迭代标题行：

```
let message = "To: jimb\r\n\
               From: superego <editor@oreilly.com>\r\n\
               \r\n\
               Did you get any writing done today?\r\n\
               When will you stop wasting time plotting fractals?\r\n";
for header in message.lines().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}
```

3.5.1 节介绍过，当一行字符串以斜杠结尾时，Rust 不会在字符串中包含下一行的缩进，因此字符串中每一行前面都不会有空格。这意味着 message 的第三行是空的。take_while 适配器一碰到这个空行就会终止迭代，因此以上代码只会打印出下面两行：

```
To: jimb
From: superego <editor@oreilly.com>
```

15.3.5 skip和skip_while

Iterator 特型的 skip 和 skip_while 方法是对 take 和 take_while 的补充，它们从迭代开始清除一定数量的项，或者一直清除到闭包发现一个可以接受的项，然后将剩余项原封不动返回。它们的签名如下：

```
fn skip(self, n: usize) -> some Iterator<Item=Self::Item>
    where Self: Sized;

fn skip_while<P>(self, predicate: P) -> some Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

skip 适配器的一个常用场景是在迭代程序的命令行参数时跳过命令名。第 2 章的最大公约数计算器中使用了以下代码遍历其命令行参数：

```
for arg in std::env::args().skip(1) {
    ...
}
```

std::env::args 函数返回一个迭代器，该迭代器会产生 String 类型的程序参数，其中第一项是程序本身的名字。程序的名字并不是循环里要处理的字符串。在那个迭代器上调用 skip(1) 会返回一个新迭代器，第一次被调用时，这个新迭代器会清除程序名，然后产生后面所有的参数。

skip_while 适配器使用闭包来决定清除序列开头的多少项。比如，可以像下面这样迭代上一节中的邮件正文：

```
for body in message.lines()
    .skip_while(|l| !l.is_empty())
    .skip(1) {
    println!("{}" , body);
}
```

这里使用 skip_while 跳过非空的行，但此时的迭代器还会产生空行，因为这个闭包碰到空行会返回 false。所以此处又使用 skip 方法跳过这个空行，这样得到的迭代器第一项就是邮件正文的第一行。对于上一节中的 message，上面的代码会打印出以下内容：

```
Did you get any writing done today?
When will you stop wasting time plotting fractals?
```

15.3.6 peekable

Iterator 特型的 peekable 方法可以让代码在不消费下一项的情况下探测下一项。调用这个方法可以将几乎任何迭代器转换为可探测的迭代器：

```
fn peekable(self) -> std::iter::Peekable<Self>
    where Self: Sized;
```

这里，Peekable<Self> 是一个实现了 Iterator<Item=Self::Item> 的结构体，而 Self 是底层迭代器的类型。

Peekable 迭代器有一个 peek 方法，该方法返回 Option<&Item>：如果底层迭代器终止就返回 None，否则返回 Some(r)，其中 r 是下一项的共享引用。（注意，如果迭代器的项类型已经是某个值的引用，那就是引用的引用。）

调用 peek 会尝试从底层迭代器取出下一项，如果取到了，就将其缓存到下一次调用 next。Peekable 上的其他 Iterator 方法都知道这个缓存。例如可探测迭代器 iter 上的 iter.last() 在耗尽底层迭代器之后知道检查缓存。

如果一开始并不知道要消费某个迭代器多少项，直到消费过程中才了解，那么可探测迭代器是非常必要的。例如，要从一个字符流中解析数值，在发现其后面第一个非数值字符之前无法确定数值是否结束：

```
use std::iter::Peekable;

fn parse_number<I>(tokens: &mut Peekable<I>) -> u32
    where I: Iterator<Item=char>
{
    let mut n = 0;
    loop {
        match tokens.peek() {
            Some(r) if r.is_digit(10) => {
                n = n * 10 + r.to_digit(10).unwrap();
            }
            _ => return n
        }
        tokens.next();
    }
}

let mut chars = "226153980,1766319049".chars().peekable();
assert_eq!(parse_number(&mut chars), 226153980);
// 注意，parse_number没有消费逗号！因此要手工消费掉
assert_eq!(chars.next(), Some(','));
assert_eq!(parse_number(&mut chars), 1766319049);
assert_eq!(chars.next(), None);
```

这个 parse_number 函数使用 peek 检查下一个字符，只有该字符是数字时才消费它。如果不是数字或者迭代器被耗尽（即 peek 返回 None），则返回已解析的数值并把下一个字符留在迭代器中供后面消费。

15.3.7 fuse

Iterator 特型并未规定迭代器返回 None 的情况下再调用 next 应该怎么做。大多数迭代器会再次返回 None，但并非全都如此。如果你的代码依靠这个行为，那可能会遇到意外。

fuse 适配器可以将任何适配器转换为第一次返回 None 之后始终继续返回 None 的迭代器。

```
struct Flaky(bool);

impl Iterator for Flaky {
    type Item = &'static str;
    fn next(&mut self) -> Option<Self::Item> {
        if self.0 {
            self.0 = false;
            Some("totally the last item")
        } else {
            self.0 = true; // 又来！
            None
        }
    }
}

let mut flaky = Flaky(true);
assert_eq!(flaky.next(), Some("totally the last item"));
assert_eq!(flaky.next(), None);
assert_eq!(flaky.next(), Some("totally the last item"));
```

```
let mut not_flaky = Flaky(true).fuse();
assert_eq!(not_flaky.next(), Some("totally the last item"));
assert_eq!(not_flaky.next(), None);
assert_eq!(not_flaky.next(), None);
```

fuse 适配器最适合需要处理不确定来源迭代器的泛型代码。这时候不用假设所有迭代器都行为一致，可以使用 fuse 确保这一点。

15.3.8　可逆迭代器与rev

有些迭代器可以从序列两端取得项。可以使用 rev 适配器反转这种迭代器。例如，一个迭代向量的迭代器可以转换为从向量末尾开始取值。这种迭代器可以实现 std::iter::DoubleEndedIterator 特型，该特型扩展了 Iterator：

```
trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

可以将这种双向迭代器想象为用两根手指标记序列的当前头和尾。从任何一端取值都会导致一根手指向另一根手指靠拢。当两根手指相遇时，迭代结束：

```
use std::iter::DoubleEndedIterator;

let bee_parts = ["head", "thorax", "abdomen"];

let mut iter = bee_parts.iter();
assert_eq!(iter.next(),      Some(&"head"));
assert_eq!(iter.next_back(), Some(&"abdomen"));
assert_eq!(iter.next(),      Some(&"thorax"));

assert_eq!(iter.next_back(), None);
assert_eq!(iter.next(),      None);
```

基于切片的迭代器可以方便地实现这个行为，因为它实际上就是一对指针，分别指向尚未产生元素的头和尾。next 和 next_back 不过是简单地从头部或尾部取出一项而已。BTreeSet 和 BTreeMap 等有序集合的迭代器也是可以两端取值的，即它们的 next_back 方法会先取得最大的元素或条目。一般来说，只要有实用性，标准库都会提供两端迭代的能力。

不过并不是所有迭代器都可以实现两端迭代。比如，基于另一个线程返回给 Receiver 值的迭代器就无法预算接收到的最后一个值是什么。通常，要查询标准库文档来确定一个迭代器是否实现了 DoubleEndedIterator。

如果迭代器实现了 DoubleEndedIterator，则可以使用 rev 适配器将其反转：

```
fn rev(self) -> some Iterator<Item=Self>
    where Self: Sized + DoubleEndedIterator;
```

返回的迭代器同样支持两端取值，其 next 和 next_back 方法只是简单地互换了一下：

```
let meals = ["breakfast", "lunch", "dinner"];

let mut iter = meals.iter().rev();
```

```
assert_eq!(iter.next(), Some(&"dinner"));
assert_eq!(iter.next(), Some(&"lunch"));
assert_eq!(iter.next(), Some(&"breakfast"));
assert_eq!(iter.next(), None);
```

在应用给可逆迭代器之后，大多数迭代器适配器会返回另一个可逆迭代器。例如，`map` 和 `filter` 都具有可逆性。

15.3.9　inspect

`inspect` 适配器可以方便地用于迭代器适配器管道的调试，但在产品代码中用得不多。这个适配器只是简单地对每一项的共享引用应用一个闭包，然后再产生相应的项。闭包不影响产生的项，但可以打印项或对项执行断言。

下面的例子展示了把字符串转换为大写会改变其长度：

```
let upper_case: String = "große".chars()
    .inspect(|c| println!("before: {:?}", c))
    .flat_map(|c| c.to_uppercase())
    .inspect(|c| println!(" after:     {:?}", c))
    .collect();
assert_eq!(upper_case, "GROSSE");
```

把小写德语字母“ß”转换为大写后是“SS”，这也是 `char::to_uppercase` 会返回字符迭代器的原因，因为大小写字母的数量并非一一对应的。前面的代码使用 `flat_map` 把 `to_uppercase` 返回的所有序列都拼接为一个 `String`，然后将它们打印出来：

```
before: 'g'
 after:     'G'
before: 'r'
 after:     'R'
before: 'o'
 after:     'O'
before: 'ß'
 after:     'S'
 after:     'S'
before: 'e'
 after:     'E'
```

15.3.10　chain

`chain` 适配器会将一个迭代器添加到另一个适配器后面。更具体地说，`i1.chain(i2)` 会返回一个迭代器，该迭代器先从 `i1` 中提取项，取完后再继续从 `i2` 中提取项。

`chain` 适配器的签名如下：

```
fn chain<U>(self, other: U) -> some Iterator<Item=Self::Item>
    where Self: Sized, U: IntoIterator<Item=Self::Item>;
```

换句话说，可以将迭代器与任何产生相同项类型的可迭代类型连缀在一块。例如：

```
let v: Vec<i32> = (1..4).chain(vec![20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

如果连缀的两个迭代器都是可逆的，则 chain 返回的迭代器也是可逆的：

```
let v: Vec<i32> = (1..4).chain(vec![20, 30, 40]).rev().collect();
assert_eq!(v, [40, 30, 20, 3, 2, 1]);
```

chain 迭代器会跟踪底层的迭代器是否返回 None，根据情况调用某个底层迭代器的 next 和 next_back。

15.3.11 enumerate

Iterator 特型的 enumerate 适配器可以向序列中添加连续的索引。对于产生项为 A,B,C... 的迭代器，其返回的迭代器则产生 (0, A),(1, B),(2, C)... 乍一看这没什么用，但实际上用处很大。

消费者可以通过索引区别不同的项，从而建立处理每一项的上下文。例如，第 2 章示例的曼德布洛特集合绘图器将图像切分成 8 个水平的长条，并把每个长条分别分派给一个不同的线程。代码中使用 enumerate 告诉每个线程其长条对应图像的哪一部分。

从矩形的像素缓冲区开始：

```
let mut pixels = vec![0; columns * rows];
```

然后使用 chunks_mut 将图像切分成水平长条，每个线程分派一个：

```
let threads = 8;
let band_rows = rows / threads + 1;
...
let bands: Vec<&mut [u8]> = pixels.chunks_mut(band_rows * columns).collect();
```

再迭代这些长条，为每个长条启动一个线程：

```
for (i, band) in bands.into_iter().enumerate() {
    let top = band_rows * i;
    // 启动一个线程渲染top..top + band_rows行
}
```

这里每次迭代都得到一对值 (i, band)，其中 band 是线程应该渲染的像素缓冲的 &mut [u8] 切片，而 i 是相应长条在整个图像中的索引。这个索引是拜本节的 enumerate 适配器所赐。有了绘图的边界和长条大小，每个线程就会知道分派给自己的是图像的哪一部分，也就知道了要把什么绘制到 band。

15.3.12 zip

zip 适配器将两个适配器组合为一个适配器，产生之前两个迭代器项的项对，就像拉链把分开的两边拼在一起一样。

例如，可以像下面这样将半开范围 0.. 与其他迭代器组合在一起，得到与使用 enumerate 适配器同样的效果：

```
let v: Vec<_> = (0..).zip("ABCD".chars()).collect();
assert_eq!(v, vec![(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]);
```

从这个意义上讲，可以把 zip 看成通用化的 enumerate。enumerate 只能给其他序列添加索引，而 zip 可以添加任何迭代项。前面提到 enumerate 可以为处理产生项提供上下文，而 zip 是达成相同目的更灵活的方式。

传给 zip 的参数不一定是迭代器本身，可以是任何可迭代类型：

```
use std::iter::repeat;

let endings = vec!["once", "twice", "chicken soup with rice"];
let rhyme: Vec<_> = repeat("going")
    .zip(endings)
    .collect();
assert_eq!(rhyme, vec![("going", "once"),
                       ("going", "twice"),
                       ("going", "chicken soup with rice")]);
```

15.3.13 by_ref

本节前面介绍的都是把适配器应用到迭代器。那么应用之后，还能再取消适配器吗？通常情况下不能，因为适配器会取得底层迭代器的所有权，没有办法再退回去了。

迭代器的 by_ref 方法可以借用迭代器的一个可修改引用，以便我们能够把适配器应用给这个引用。在通过适配器消费完迭代器的项之后，借用结束，恢复对原始迭代器的访问。

例如，本章前面展示了使用 take_while 和 skip_while 处理邮件的标题行和正文。如果想基于同一个底层迭代器完成这两个操作该怎么办？使用 by_ref 可以让 take_while 处理标题行，处理完之后，再回到底层迭代器，take_while 剩下的正好是邮件正文：

```
let message = "To: jimb\r\n\
               From: id\r\n\
               \r\n\
               Oooooh, donuts!!\r\n";

let mut lines = message.lines();

println!("Headers:");
for header in lines.by_ref().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}

println!("\nBody:");
for body in lines {
    println!("{}" , body);
}
```

lines.by_ref() 调用从迭代器借用了一个可修改引用，而 take_while 迭代器只是取得了这个引用的所有权。第一个 for 循环结束时，take_while 迭代器离开作用域，借用关系随之终止。因此，又可以在第二个 for 循环中继续使用 lines 了。打印的最终结果如下：

```
Headers:
To: jimb
From: id
```

```
Body:
Oooooh, donuts!!
```

by_ref 适配器的定义很简单，就是返回迭代器的可修改引用。然后，标准库就包含了下面这个奇怪的小实现：

```
impl<'a, I: Iterator + ?Sized> Iterator for &'a mut I {
    type Item = I::Item;
    fn next(&mut self) -> Option<I::Item> {
        (**self).next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        (**self).size_hint()
    }
}
```

换句话说，如果 I 是某种迭代器类型，那么 &mut I 也是迭代器，其 next 和 size_hint 方法会解引用到它的引用值。在对迭代器的可修改引用调用适配器时，适配器取得**引用**而非迭代器本身的所有权。这就是在适配器超出作用域时会终止的一个借用关系。

15.3.14 cloned

cloned 适配器将一个产生引用的迭代器转换为产生基于引用克隆的值的迭代器。自然地，引用值的类型必须实现 Clone。例如：

```
let a = ['1', '2', '3', '∞'];

assert_eq!(a.iter().next(),          Some(&'1'));
assert_eq!(a.iter().cloned().next(), Some('1'));
```

15.3.15 cycle

cycle 适配器返回一个无休止重复底层迭代器的迭代器。底层迭代器必须实现 std::clone::Clone，以便 cycle 可以保存其初始状态并在每次循环开始时重用。例如：

```
let dirs = ["North", "East", "South", "West"];
let mut spin = dirs.iter().cycle();
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
assert_eq!(spin.next(), Some(&"South"));
assert_eq!(spin.next(), Some(&"West"));
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
```

再看另一个使用迭代器的例子：

```
use std::iter::{once, repeat};

let fizzes = repeat("").take(2).chain(once("fizz")).cycle();
let buzzes = repeat("").take(4).chain(once("buzz")).cycle();
let fizzes_buzzes = fizzes.zip(buzzes);
```

```
let fizz_buzz = (1..100).zip(fizzes_buzzes)
    .map(|tuple|
         match tuple {
             (i, ("", "")) => i.to_string(),
             (_, (fizz, buzz)) => format!("{}{}", fizz, buzz)
         });

for line in fizz_buzz {
    println!("{}", line);
}
```

这本来是一个小孩子的文字游戏，但现在有时候会用作程序员的面试题。玩家要通过计算，用“fizz”替换可以被 3 整除的数，用“buzz”替换可以被 5 整除的数。可以同时被两者整除的数就会得到“fizzbuzz”。

15.4　消费迭代器

目前为止，本书讨论的都是创建迭代器，以及通过适配器将它们变成新迭代器。本节开始介绍消费迭代器。

当然，使用 `for` 循环也可以消费迭代器，直接调用 `next` 也行。但是，有很多常见的任务不需要每次都这样重复地写出来。`Iterator` 特型提供了很多方法，可以直接完成这些任务。

15.4.1　简单累计：count、sum和product

`count` 方法从一个迭代器中取值，直到它返回 `None`，然后告诉你这个迭代器包含多少项。下面这个示例计算标准输入的行数：

```
use std::io::prelude::*;

fn main() {
    let stdin = std::io::stdin();
    println!("{}", stdin.lock().lines().count());
}
```

`sum` 和 `product` 方法分别用于计算迭代器项的和与积。当然，迭代器项必须是整数或浮点数：

```
fn triangle(n: u64) -> u64 {
    (1..n+1).sum()
}
assert_eq!(triangle(20), 210);

fn factorial(n: u64) -> u64 {
    (1..n+1).product()
}
assert_eq!(factorial(20), 2432902008176640000);
```

（通过实现 `std::iter::Sum` 和 `std::iter::Product` 特型可以扩展 `sum` 和 `product` 方法，以处理其他类型。本书没有介绍这两个特型。）

15.4.2　max和min

Iterator 上的 max 和 min 方法分别返回迭代器产生项的最大值和最小值。迭代器的项类型必须实现 std::cmp::Ord，这样项与项之间才能比较。例如：

```
assert_eq!([-2, 0, 1, 0, -2, -5].iter().max(), Some(&1));
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));
```

这两个方法返回 Option<Self::Item>，因此在迭代器没有产生项时返回 None。

正如 12.2 节所解释的，Rust 的浮点类型 f32 和 f64 只实现了 std::cmp::PartialOrd，没有实现 std::cmp::Ord。因此不能使用 max 和 min 比较浮点值序列的最大值和最小值。Rust 这种不迎合大众口味的设计是有意的，因为该如何处理 IEEE 的 NaN 值这些函数并不确定。如果简单地忽略它们则可能在代码中引入更严重的问题。

如果你知道如何处理 NaN 值，那么可以使用 max_by 和 min_by 迭代器方法。这两个方法支持提供自定义比较方法。

15.4.3　max_by和min_by

max_by 和 min_by 方法根据提供的自定义比较方法返回迭代器产生的最大值和最小值：

```
use std::cmp::{PartialOrd, Ordering};

// 比较两个f64值。如果包含NaN则诧异
fn cmp(lhs: &&f64, rhs: &&f64) -> Ordering {
    lhs.partial_cmp(rhs).unwrap()
}

let numbers = [1.0, 4.0, 2.0];
assert_eq!(numbers.iter().max_by(cmp), Some(&4.0));
assert_eq!(numbers.iter().min_by(cmp), Some(&1.0));

let numbers = [1.0, 4.0, std::f64::NAN, 2.0];
assert_eq!(numbers.iter().max_by(cmp), Some(&4.0));  // 诧异
```

（这里 cmp 参数的双重引用是因为 numbers.iter() 产生对元素的引用，然后 max_by 和 min_by 又把迭代器项的引用传给闭包。）

15.4.4　max_by_key和min_by_key

Iterator 上的 max_by_key 和 min_by_key 方法可以根据应用到每一项的闭包选择最大或最小的项。闭包可以选择项的某个字段，或者对每一项执行某些计算。我们通常并不关心这些极值本身，而只关心与最大项或最小项关联的数据。因此，这两个方法通常比 max 和 min 也更有用。它们的签名如下：

```
fn min_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;

fn max_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;
```

这也就是说，对给定的以每一项作为参数且返回有序类型 B 的闭包，它们返回闭包返回的 B 最大或最小的项。如果没有产生项，则返回 None。

例如，如果要从一个城市的散列表中找到其中人口最多和最少的城市，可以这样做：

```
use std::collections::HashMap;

let mut populations = HashMap::new();
populations.insert("Portland",  583_776);
populations.insert("Fossil",        449);
populations.insert("Greenhorn",       2);
populations.insert("Boring",      7_762);
populations.insert("The Dalles", 15_340);

assert_eq!(populations.iter().max_by_key(|&(_name, pop)| pop),
           Some((&"Portland", &583_776)));
assert_eq!(populations.iter().min_by_key(|&(_name, pop)| pop),
           Some((&"Greenhorn", &2)));
```

这里的闭包 |&(_name, pop)| pop 会应用于迭代器产生的每一项，返回的值则用于比较。在这个例子中返回的是人口，因此就会比较人口。返回的值是整个项，而不是闭包返回的值。（自然地，如果需要频繁进行这样的查询，那么可能需要找到一种更有效的查询方式，而不是像这样对散列表执行线性搜索。）

15.4.5　比较项序列

可以使用 < 和 == 操作符比较字符串、向量和切片，假设它们的每个元素都是可以比较的。尽管迭代器不支持 Rust 的比较操作符，但它们提供了 eq 和 lt 等方法以实现同样的功能。这些方法从迭代器中取得成对的项，然后对它们进行比较，直到可以做出决定。例如：

```
let packed =  "Helen of Troy";
let spaced =  "Helen   of    Troy";
let obscure = "Helen of Sandusky"; // 不错的人，只是名气不大

assert!(packed != spaced);
assert!(packed.split_whitespace().eq(spaced.split_whitespace()));

// 返回true，因为' ' < 'o'
assert!(spaced < obscure);

// 返回true，因为'Troy' > 'Sandusky'
assert!(spaced.split_whitespace().gt(obscure.split_whitespace()));
```

调用 split_whitespace 返回的迭代器产生以空格分割字符串得到的单词。调用迭代器的 eq 和 gt 方法可以执行单词与单词的比较，而不是字符与字符的比较。之所以可以这样，是因为 &str 实现了 PartialOrd 和 PartialEq。

迭代器既提供了 eq 和 ne 方法用于相等性比较，也提供了 lt、le、gt 和 ge 方法用于次序比较。cmp 和 partial_cmp 方法与 Ord 和 PartialOrd 特型上对应的方法类似。

15.4.6　any和all

any 和 all 方法给迭代器产生的每一项应用闭包，如果闭包对其中一项或全部项返回 true，则返回 true：

```
let id = "Iterator";

assert!( id.chars().any(char::is_uppercase));
assert!(!id.chars().all(char::is_uppercase));
```

这些方法只消费确定答案必要的项。例如，如果闭包对某一项返回 true，那 any 就立即返回 true，而不会再从迭代器中取更多项。

15.4.7　position、rposition和ExactSizeIterator

position 方法给迭代器产生的每一项应用闭包，返回闭包返回 true 的第一项的索引。更精确地说，position 返回一个索引的 Option：如果闭包对任何项都没有返回 true，则返回 None。只要闭包返回 true，position 就停止取值。例如：

```
let text = "Xerxes";
assert_eq!(text.chars().position(|c| c == 'e'), Some(1));
assert_eq!(text.chars().position(|c| c == 'z'), None);
```

rposition 方法与 position 方法相同，只是其从右侧进行搜索。例如：

```
let bytes = b"Xerxes";
assert_eq!(bytes.iter().rposition(|&c| c == b'e'), Some(4));
assert_eq!(bytes.iter().rposition(|&c| c == b'X'), Some(0));
```

rposition 方法要求使用可逆迭代器，这样才能从序列右端取值。另外，它也要求迭代器大小固定，这样才能像 position 那样为索引赋值，以最左边的项为 0。固定大小迭代器是指实现 std::iter::ExactSizeIterator 特型的迭代器：

```
pub trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

其中，len 方法返回剩余项数，而 is_empty 方法在迭代完成时返回 true。

当然，并不是所有迭代器都能提前知道自己会产生多少项，在前面的例子中，产生 &str 的 chars 迭代器就不知道（UTF-8 是变长编码）。因此不能对字符串使用 rposition。但产生字节数组的迭代器知道数组长度，因此可以实现 ExactSizeIterator。

15.4.8　fold

fold 方法是一个通用工具，可以对迭代器产生项的整个序列执行某些累计操作。这个方法接收一个名为**累加器**（accumulator）的初始值和一个闭包，然后对当前累加器和迭代器的下一项重复应用闭包。每次闭包的返回值都会成为累加器的新值，然后再和迭代器的下一项一块传给闭包。累加器的最终值也是 fold 方法返回的值。如果序列是空的，则 fold 返

回累加器的初始值。

消费迭代器值的许多其他方法可以写成使用 fold 的形式：

```
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n+1), 6);        // count
assert_eq!(a.iter().fold(0, |n, i| n+i), 45);       // sum
assert_eq!(a.iter().fold(1, |n, i| n*i), 151200);   // product

// max
assert_eq!(a.iter().fold(i32::min_value(), |m, &i| std::cmp::max(m, i)),
           10);
```

fold 方法的签名如下：

```
fn fold<A, F>(self, init: A, f: F) -> A
    where Self: Sized, F: FnMut(A, Self::Item) -> A;
```

这里，A 是累加器类型。init 参数是一个 A，闭包的第一个参数及返回值，还有 fold 本身的返回值都是 A。

注意，累加器的值会转移到闭包中再转移出来，因此可以对非 Copy 类型的累加器使用 fold：

```
let a = ["Pack ", "my ", "box ", "with ",
         "five ", "dozen ", "liquor ", "jugs"];

let pangram = a.iter().fold(String::new(),
                            |mut s, &w| { s.push_str(w); s });
assert_eq!(pangram, "Pack my box with five dozen liquor jugs");
```

15.4.9 nth

nth 方法接收一个索引值 n，然后跳过迭代器中相应的项，返回索引对应的项；如果序列在此之前结束，则返回 None。调用 .nth(0) 等同于调用 .next()。

nth 不会像适配器那样取得迭代器的所有权，因此可以多次调用：

```
let mut squares = (0..10).map(|i| i*i);

assert_eq!(squares.nth(4), Some(16));
assert_eq!(squares.nth(0), Some(25));
assert_eq!(squares.nth(6), None);
```

这个方法的签名如下：

```
fn nth(&mut self, n: usize) -> Option<Self::Item>
    where Self: Sized;
```

15.4.10 last

last 方法消费每一项，直到迭代器返回 None，然后返回最后一项。如果迭代器不产生任何项，则 last 返回 None。它的签名如下：

```
fn last(self) -> Option<Self::Item>;
```

例如：

```
let squares = (0..10).map(|i| i*i);
assert_eq!(squares.last(), Some(81));
```

这样会从前面开始消费迭代器的所有项，即使迭代器是可逆的。因此，如果迭代器是可逆的，又不需要消费其所有项，则可以只写成 `iter.rev().next()`。

15.4.11 find

`find` 方法从迭代器中取得第一个闭包返回 `true` 的值，或者如果没有找到合适的项则返回 `None`。它的签名如下：

```
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
    where Self: Sized,
          P: FnMut(&Self::Item) -> bool;
```

例如，使用 15.4.4 节中城市和人口的散列表，可以这样查找：

```
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 1_000_000),
           None);
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 500_000),
           Some((&"Portland", &583_776)));
```

表中没有人口超过 100 万的城市，但有一个城市人口超过了 50 万。

15.4.12 构建集合：collect和FromIterator

本书已经多次使用 `collect` 方法来构建保存迭代器项的向量。例如，第 2 章调用 `std::env::args()` 取得了一个程序命令行参数的迭代器，然后调用这个迭代器的 `collect` 方法将它们汇集为了一个向量：

```
let args: Vec<String> = std::env::args().collect();
```

但 `collect` 方法并不只是针对向量。事实上，它可以用来构建 Rust 标准库中的任何集合，只要迭代器产生类型合适的项就行：

```
use std::collections::{HashSet, BTreeSet, LinkedList, HashMap, BTreeMap};

let args: HashSet<String> = std::env::args().collect();
let args: BTreeSet<String> = std::env::args().collect();
let args: LinkedList<String> = std::env::args().collect();

// 汇集映射需要(key, value)对，因此对这个例子而言，
// 可以使用zip为字符串添加整数索引
let args: HashMap<String, usize> = std::env::args().zip(0..).collect();
let args: BTreeMap<String, usize> = std::env::args().zip(0..).collect();

// 更多例子
```

自然地，`collect` 本身并不知道如何构建这些类型。相反地，`Vec` 或 `HashMap` 等集合类型知道如何基于迭代器构建自己，所以它们实现了 `std::iter::FromIterator` 特型，而 `collect`

只是一个快捷方式而已：

```
trait FromIterator<A>: Sized {
    fn from_iter<T: IntoIterator<Item=A>>(iter: T) -> Self;
}
```

如果某个集合类型实现了 FromIterator<A>，那么其静态方法 from_iter 就可以基于产生类型 A 的可迭代类型构建该类型的值。

在最简单的情况下，实现可以简单地构建一个空集合，然后逐个将迭代器产生的项添加到这个集合中。例如，std::collections::LinkedList 对 FromIterator 的实现就是这样做的。

不过，某些类型的实现方式可以更好一些。例如，基于迭代器 iter 构建一个向量可以像下面这么简单：

```
let mut vec = Vec::new();
for item in iter {
    vec.push(item)
}
vec
```

但这并不理想。随着向量增长，可能需要扩展其缓冲区，要求调用堆分配器以及复制已有元素。向量确实在算法上考虑到了保持这个过程的开销最低，但如果有某种方式可以简单地将初始缓冲区一开始就分配为合适大小，则无须再调用大小了。

这也是 Iterator 特型有一个 size_hint 方法的原因：

```
trait Iterator {
    ...
    fn size_hint(&self) -> (usize, Option<usize>) {
        (0, None)
    }
}
```

size_hint 方法返回迭代器产生项数的下限值和可选的上限值。默认定义是返回 0 作为下限值，不给出上限值，相当于说“不知道”。但很多迭代器可以对此给出更有用的信息。例如，产生 Range 的迭代器就知道自己要产生多少值，产生 Vec 和 HashMap 的迭代器也一样。这些迭代器都提供了自己的对 size_hint 的特殊定义。

这两个边界值正是 Vec 对 FromIterator 的实现从一开始就为新向量缓冲区分配正确大小所必需的信息。插入值时仍然会检查缓冲区是否够大，因此即使这个提示信息不正确，也只会影响性能，而不会影响安全。其他类型也可以如法炮制。例如，HashSet 和 HashMap 也会使用 Iterator::size_hint 为自己的散列表创建适当的初始大小。

这里有一个关于类型推断的问题。在前面的例子中，同样是调用 std::env::args().collect() 却会根据上下文产生 4 种不同的集合。这看起来可能有点奇怪。实际上，collect 的返回类型是其类型参数，因此前两个调用相当于：

```
let args = std::env::args().collect::<vec<String>>();
let args = std::env::args().collect::<HashSet<String>>();
```

不过，哪怕只有一个适合 collect 参数的类型，Rust 的类型推断都会提供出来。像这样明

确写出 args 的参数，可以保证类型正确。

15.4.13　Extend特型

如果类型实现了 std::iter::Extend 特型，那么它的 extend 方法可以将一个迭代器的项添加到集合中：

```
let mut v: Vec<i32> = (0..5).map(|i| 1 << i).collect();
v.extend(&[31, 57, 99, 163]);
assert_eq!(v, &[1, 2, 4, 8, 16, 31, 57, 99, 163]);
```

所有的标准库集合都实现了 Extend，因此它们都支持这个方法，String 也一样。但固定长度的数组和切片不支持这个方法。

这个特型的定义如下所示：

```
trait Extend<A> {
    fn extend<T>(&mut self, iter: T)
        where T: IntoIterator<Item=A>;
}
```

显然，这跟 std::iter::FromIterator 非常相似。std::iter::FromIterator 创建一个新集合，而 Extend 扩展现有集合。事实上，标准库中的一些 FromIterator 实现就是先创建一个新的空集合，然后再调用 extend 填充这个集合。例如，std::collections::LinkedList 是这样实现 FromIterator 的：

```
impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}
```

15.4.14　partition

partition 方法把一个迭代器的项分成两个集合，然后使用闭包决定哪一项属于哪个集合：

```
let things = ["doorknob", "mushroom", "noodle", "giraffe", "grapefruit"];

// 震惊：凡是有生命的，其名字的首字母始终是奇数个字母
let (living, nonliving): (Vec<&str>, Vec<&str>)
    = things.iter().partition(|name| name.as_bytes()[0] & 1 != 0);

assert_eq!(living,    vec!["mushroom", "giraffe", "grapefruit"]);
assert_eq!(nonliving, vec!["doorknob", "noodle"]);
```

与 collect 类似，partition 可以生成任何类型的集合（当然两个集合必须是相同类型）。但与 collect 不同，这里需要指定返回类型：前面的例子写出了 living 和 nonliving 的类型，让类型推断可以选择调用正确的类型参数。

partition 的签名如下：

```
fn partition<B, F>(self, f: F) -> (B, B)
    where Self: Sized,
          B: Default + Extend<Self::Item>,
          F: FnMut(&Self::Item) -> bool;
```

collect 要求其结果类型实现 FromIterator，partition 则要求其结果类型实现 std::default::Default（对此，所有 Rust 集合的实现都是返回一个空集合）和 std::default::Extend。

partition 为什么不把迭代器分成两个迭代器，而是两个集合呢？一个原因是从底层迭代器中取出但还没有从切分后的迭代器中取出的值需要先缓存在某个地方，而这最终可能需要在内部先构建某种集合。但更本质的原因是这跟安全相关。把一个迭代器切成两个需要两部分共享同一个底层迭代器，而迭代器必须可以修改才能使用。底层迭代器必须共享且可修改对 Rust 来说是违背安全准则的。

15.5　实现自己的迭代器

自定义类型也可以实现 IntoIterator 和 Iterator 特型，从而可以让本章介绍的所有适配器和消费者都派上用场，当然还包括使用标准迭代器接口的第三方库。本节先介绍基于一个范围类型实现简单的迭代器，再介绍基于一个二叉树类型实现较复杂的迭代器。

假设有如下范围类型（在标准库 std::ops::Range<T> 类型基础上做了简化）：

```
struct I32Range {
    start: i32,
    end: i32
}
```

迭代 I32Range 需要两个状态：当前值和结束迭代的限制条件。对于 I32Range 而言，可以使用 start 作为下一个值，使用 end 作为限制条件。因此，可以像下面这样实现 Iterator：

```
impl Iterator for I32Range {
    type Item = i32;
    fn next(&mut self) -> Option<i32> {
        if self.start >= self.end {
            return None;
        }
        let result = Some(self.start);
        self.start += 1;
        result
    }
}
```

这个迭代器产生 i32 类型的项，因此 i32 就是 Item 的类型。如果迭代完成，那么 next 返回 None；否则，产生下一个值并更新状态以备下次调用。

当然，for 循环要使用 IntoIterator::into_iter 将其操作数转换为一个迭代器。但标准库为每个实现 Iterator 的类型都提供了对 IntoIterator 的通用实现，因此 I32Range 已经可以使用了：

```
let mut pi = 0.0;
let mut numerator = 1.0;

for k in (I32Range { start: 0, end: 14 }) {
    pi += numerator / (2*k + 1) as f64;
    numerator /= -3.0;
}
pi *= f64::sqrt(12.0);

// IEEE 754明确定义了这个结果
assert_eq!(pi as f32, std::f32::consts::PI);
```

不过 I32Range 有点特殊，它的可迭代类型和迭代器都是同一种类型。很多情况并没有那么简单。例如，下面是第 10 章出现过的二叉树类型：

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>
}
```

遍历二叉树的经典方式是递归，通过函数调用栈跟踪树中的位置和还未访问的节点。但在为 BinaryTree<T> 实现 Iterator 时，每次调用 next 必须实际产生并返回一个值。为跟踪还未产生的树节点，迭代器必须维护自己的栈。下面是针对 BinaryTree 的一个可能的迭代器类型：

```
use self::BinaryTree::*;

// BinaryTree的按序遍历状态
struct TreeIter<'a, T: 'a> {
    // 树节点引用的栈。 因为使用的是Vec的push和pop方法，
    // 所以栈顶是向量的末尾
    //
    // 节点迭代器将从栈顶取得下一个值，未访问的祖先节点在下面。
    // 如果栈空了，则迭代结束
    unvisited: Vec<&'a TreeNode<T>>
}
```

常见的操作是先将子树左边的节点推到栈里，为此，在 TreeIter 上定义这么一个方法：

```
impl<'a, T: 'a> TreeIter<'a, T> {
    fn push_left_edge(&mut self, mut tree: &'a BinaryTree<T>) {
        while let NonEmpty(ref node) = *tree {
            self.unvisited.push(node);
            tree = &node.left;
        }
    }
}
```

有了这个辅助方法，可以再为 BinaryTree 定义一个 iter 方法，返回树的迭代器：

```
impl<T> BinaryTree<T> {
    fn iter(&self) -> TreeIter<T> {
        let mut iter = TreeIter { unvisited: Vec::new() };
        iter.push_left_edge(self);
        iter
    }
}
```

这个 iter 方法构建了一个空 TreeIter，然后调用 push_left_edge 来填充初始的栈。按照 unvisited 栈的规则，最左边的节点在上面。

按照标准库的通行做法，接下来可以在树的一个共享引用上实现 IntoIterator，调用 BinaryTree::iter 返回迭代器：

```
impl<'a, T: 'a> IntoIterator for &'a BinaryTree<T> {
    type Item = &'a T;
    type IntoIter = TreeIter<'a, T>;
    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}
```

这个 IntoIter 定义将 TreeIter 作为 &BinaryTree 的迭代器类型。

最后，在 Iterator 的实现中，要实际地遍历树。与 BinaryTree 的 iter 方法类似，迭代器的 next 方法同样按照栈的规则行事：

```
impl<'a, T> Iterator for TreeIter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<&'a T> {
        // 查找迭代必须产生的节点，或者结束迭代
        let node = match self.unvisited.pop() {
            None => return None,
            Some(n) => n
        };

        // 当前节点后面的下一个节点是当前节点右子节点的最左子节点，
        // 因此把它推进栈里
        self.push_left_edge(&node.right);

        // 产生对节点值的引用
        Some(&node.element)
    }
}
```

如果栈空了，则迭代结束。否则，node 就是对当前节点的引用，而调用会返回对其 element 字段的引用。不过首先必须把迭代器状态推进下一个节点。如果这个节点有右子树，那么要访问的下一个节点就是这个子树的最左节点。如果这个节点没有右子树，push_left_edge 则没有效果，这也是我们想要的：可以认为新的栈顶节点是 node 的第一个未访问的祖先（如果有的话）。

有了 IntoIterator 和 Iterator 实现，就可以使用 for 循环按引用迭代 BinaryTree 了：

```
fn make_node<T>(left: BinaryTree<T>, element: T, right: BinaryTree<T>)
           -> BinaryTree<T>
```

```
{
    NonEmpty(Box::new(TreeNode { left, element, right }))
}

// 构建一棵小树
let subtree_l = make_node(Empty, "mecha", Empty);
let subtree_rl = make_node(Empty, "droid", Empty);
let subtree_r = make_node(subtree_rl, "robot", Empty);
let tree = make_node(subtree_l, "Jaeger", subtree_r);

// 迭代树
let mut v = Vec::new();
for kind in &tree {
    v.push(*kind);
}
assert_eq!(v, ["mecha", "Jaeger", "droid", "robot"]);
```

图 15-1 展示了在迭代示例中的树时 `unvisited` 栈的行为。在每一步中，要访问的下一个节点都在栈顶，而其未访问的祖先节点都在下面。

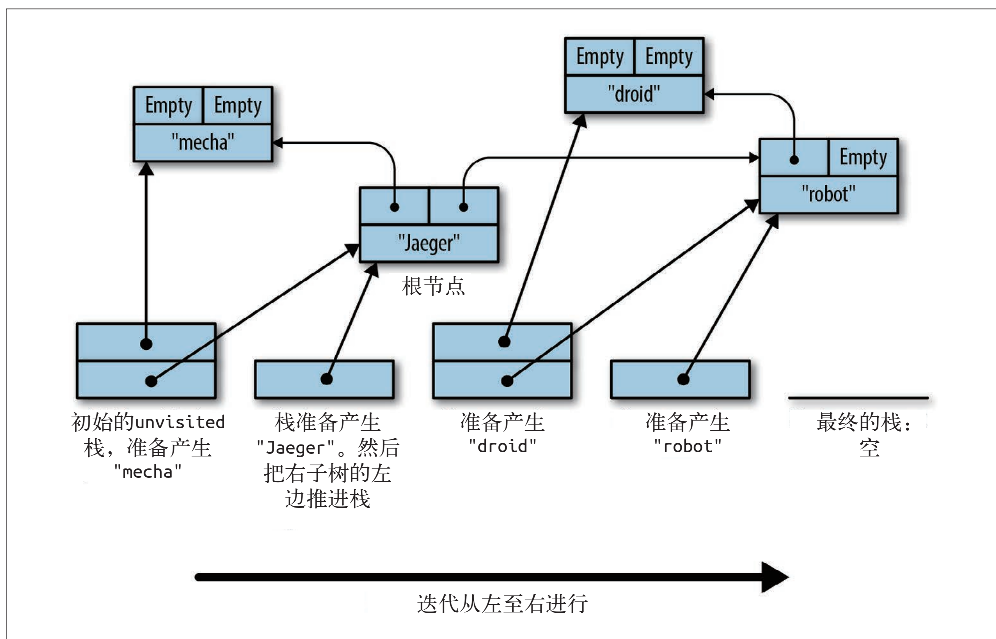


图 15-1：迭代二叉树

所有常用的迭代器适配器和消费者都可以在这个树上使用：

```
assert_eq!(tree.iter()
           .map(|name| format!("mega-{}", name))
           .collect::<Vec<_>>(),
           vec!["mega-mecha", "mega-Jaeger",
                "mega-droid", "mega-robot"]);
```

第 16 章

集合

> 我们都像麦克斯韦妖一样活动。生物体（organism），顾名思义，时刻在组织（organize）。也正是在日常经验中，我们可以发现一向冷静的物理学家之所以会在两个世纪里对这个卡通形象一直难以忘怀的原因。我们分拣邮件、堆造沙堡、拼凑拼图、复盘棋局、收集邮票、给麦穗脱粒、按字母表顺序排列图书、创造对称形式、创作十四行诗和奏鸣曲，以及整理自己的房间。所有这些活动并不需要巨大的能量，只需保障我们能够发挥智能便可。
>
> ——James Gleick，《信息简史》[1]

Rust 标准库包含几个**集合性的**泛型类型，用于在内存中存储数据。之前我们已经用过集合了，比如 `Vec` 和 `HashMap`。本章将详细介绍这两个类型以及其他几个标准集合的方法。在此之前，需要先明确一下 Rust 集合与其他语言中集合的系统性差异。

首先，转移和借用无处不在。Rust 使用转移来避免深复制值。这也是 `Vec<T>::push(item)` 按值而非按引用取得参数的原因。这样值就转移到了向量中。第 4 章中的那些图展示了这个过程。把 Rust 字符串推到 `Vec<String>` 中很快，因为 Rust 不必复制字符串的字符数据，而且字符串的所有权始终非常明确。

其次，Rust 没有无效错误，比如在集合中保存数据指针，而在集合缩放或被修改之后出现悬空指针。无效错误是 C++ 中另一个未定义行为的来源。即使在内存安全的语言中，无效错误偶尔也会导致 `ConcurrentModificationException`。Rust 借用检查器在编译时就可以排除这些问题。

最后，Rust 没有 `null`，因此在其他语言会出现 `null` 的地方 Rust 用 `Option` 代替。

注 1：此书已由人民邮电出版社出版，详见 http://ituring.cn/book/731。——编者注

除了以上差异，Rust 集合跟你想象中的应该都一样。有经验的程序员可以直接跳到 16.6.1 节。

16.1 概述

表 16-1 展示了 Rust 的 8 个标准集合，它们全部都是泛型类型。

表16-1：标准集合

集合	说明	其他语言中类似的集合		
		C++	Java	Python
Vec<T>	可增数组	vector	ArrayList	list
VecDeque<T>	双端队列 （可增长环形缓冲区）	deque	ArrayDeque	collections .deque
LinkedList<T>	双向链表	list	LinkedList	–
BinaryHeap<T> where T: Ord	最大堆	priority_queue	PriorityQueue	heapq
HashMap<K, V> where K: Eq + Hash	键 – 值散列表	unordered_map	HashMap	dict
BTreeMap<K, V> where K: Ord	有序键 – 值表	map	TreeMap	–
HashSet<T> where T: Eq + Hash	散列表	unordered_set	HashSet	set
BTreeSet<T> where T: Ord	有序集	set	TreeSet	–

Vec<T>、HashMap<K, V> 和 HashSet<T> 是最常用的集合类型，其他类型用处有限。本章将逐一讨论这些集合。

- Vec<T> 是可增长的、分配在堆上的类型 T 的值的数组。本章会用近一半篇幅介绍 Vec 及其有用的方法。
- VecDeque<T> 与 Vec<T> 类似，但适合作为先进先出队列使用。它支持在列表前端和后端高效地添加和移除值。但这样会导致其他所有操作稍微变慢一些。
- LinkedList<T> 支持在列表前、后端快速存取，与 VecDeque<T> 类似，但增加了快速连接。不过，通常来说，LinkedList<T> 还是比 Vec<T> 和 VecDeque<T> 慢一些。
- BinaryHeap<T> 是一个优先队列。BinaryHeap 中值的组织方式始终适合查找和移除最大值。
- HashMap<K, V> 是一个键 – 值对的表，按键查值很快。条目以任意顺序存储。
- BTreeMap<K, V> 与 HashMap<K, V> 类似，但条目以键的顺序存储。比如，BTreeMap <String, i32> 按字符串比较顺序存储其条目。除非需要按序存储条目，否则用 HashMap<K, V> 更快。
- HashSet<T> 是一组类型 T 的值，添加和移除值很快，检查给定值是否在集合中存在也很快。
- BTreeSet<T> 与 HashSet<T> 类似，但值是按顺序存储的。同样，除非需要保证数据的顺序，否则用 HashSet<T> 更快。

16.2　Vec<T>

之前已经多次用到过 Vec，所以假设大家对它已经熟悉了。不熟悉的可以参考 3.4.2 节。接下来我们会深入介绍它的方法和内部工作原理。

创建向量最简单的方式是使用 vec! 宏：

```
// 创建空向量
let mut numbers: Vec<i32> = vec![];

// 创建包含给定内容的向量
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024]; // 1024个填充0的字节
```

第 4 章介绍过，向量有 3 个字段：长度、容量和指向存储其元素的堆内存的指针。图 16-1 形象地展示了前面代码创建的向量在内存中的布局。空向量 numbers 的初始容量为 0。在为它添加第一个元素之前，不会分配堆内存。

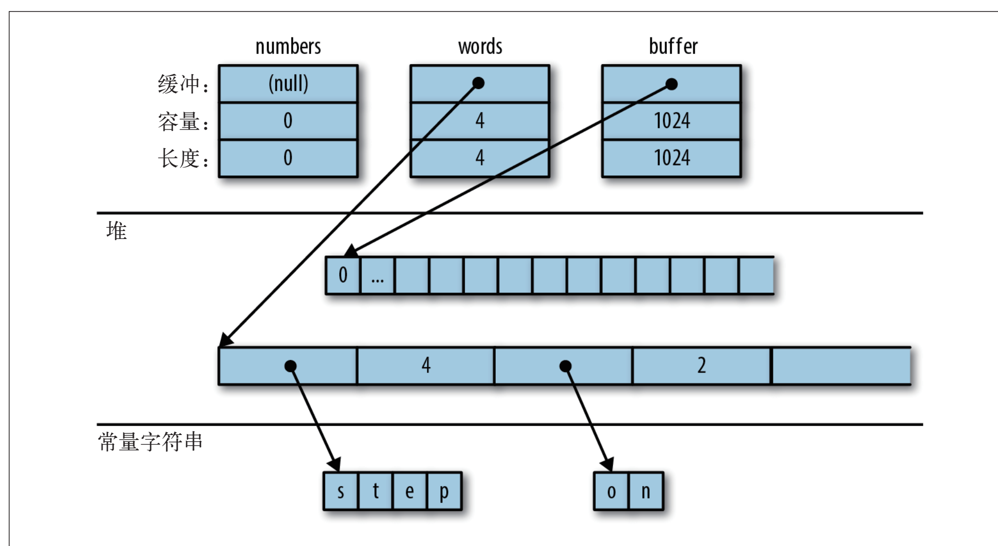


图 16-1：向量的内存布局。words 的每个元素都是一个包含指针和长度的 &str 值

与所有集合一样，Vec 实现了 std::iter::FromIterator。因此可以使用迭代器的 .collect() 方法基于任何迭代器创建向量，15.4.12 节已经介绍过了：

```
// 把另一个集合转换为向量
let my_vec = my_set.into_iter().collect::<Vec<String>>();
```

16.2.1　访问元素

通过索引可以直观地获取数组、切片或向量的元素：

```
// 取得一个元素的引用
let first_line = &lines[0];

// 取得一个元素的副本
let fifth_number = numbers[4];       // 要求Copy
let second_line = lines[1].clone(); // 要求Clone

// 取得一个切片的引用
let my_ref = &buffer[4..12];

// 取得一个切片的副本
let my_copy = buffer[4..12].to_vec(); // 要求Clone
```

如果索引越界，则所有这些形式都会诧异。

Rust 对数值类型很挑剔，对向量也不例外。向量长度和索引是 `usize` 类型。使用 `u32`、`u64` 或 `isize` 作为向量索引会出错。必要时可以使用 `n as usize` 来转换，参见 6.13 节。

以下这些方法可以方便地访问向量或切片的特定元素（注意，所有切片方法也可以在数组和向量上使用）。

- **`slice.first()`** 返回对 `slice` 第一个元素的引用（如果有的话）。

  返回值类型是 `Option<&T>`，因此如果 `slice` 为空则返回值是 `None`，如果不为空则返回 `Some(&slice[0])`。

  ```
  if let Some(item) = v.first() {
      println!("We got one! {}", item);
  }
  ```

- **`slice.last()`** 与 `slice.first()` 类似，只是返回最后一个元素的引用。
- **`slice.get(index)`** 返回 `slice[index]` 的 `Some` 引用（如果它存在的话），或者如果 `slice` 的元素少于 `index+1` 个则返回 `None`。

  ```
  let slice = [0, 1, 2, 3];
  assert_eq!(slice.get(2), Some(&2));
  assert_eq!(slice.get(4), None);
  ```

- **`slice.first_mut()`**、**`slice.last_mut()`** 和 **`slice.get_mut(index)`** 是上面几个方法的变体，返回 `mut` 引用。

  ```
  let mut slice = [0, 1, 2, 3];
  {
      let last = slice.last_mut().unwrap(); // last的类型是&mut i32
      assert_eq!(*last, 3);
      *last = 100;
  }
  assert_eq!(slice, [0, 1, 2, 100]);
  ```

因为返回 `T` 值意味着转移，所以就地访问元素的方法通常都返回元素的引用。

一个例外是 `.to_vec()` 方法，它返回元素的副本。

- **`slice.to_vec()`** 克隆整个切片，返回一个新向量。

```
let v = [1, 2, 3, 4, 5, 6, 7, 8, 9];
assert_eq!(v.to_vec(),
           vec![1, 2, 3, 4, 5, 6, 7, 8, 9]);
assert_eq!(v[0..6].to_vec(),
           vec![1, 2, 3, 4, 5, 6]);
```

这个方法只在元素是可以克隆（where T: Clone）的情况下有效。

16.2.2 迭代

向量和切片是可以迭代的，遵循 15.2.2 节的模式，既可以按值迭代，也可以按引用迭代。

- 迭代 Vec<T> 产生 T 类型的项。元素逐个从向量中转移出来并被消费。
- 迭代 &[T; N]、&[T] 或 &Vec<T> 类型的值，即对数组、切片或向量的引用，产生 &T 类型的项，即对个别元素的引用，不会转移。
- 迭代 &mut [T; N]、&mut [T] 或 &mut Vec<T> 类型的值产生 &mut T 类型的项。

数组、切片和向量还有 .iter() 和 .iter_mut() 方法（参见 15.2.1 节），这两个方法创建的迭代器产生对它们元素的引用。16.2.5 节还会介绍迭代切片的更新奇方式。

16.2.3 增长和收缩向量

数组、切片和向量的**长度**表示它们包含元素的数量。

- **slice.len()** 返回 slice 的长度，类型为 usize。
- **slice.is_empty()** 在 slice 不包含元素（即 slice.len() == 0）时返回 true。

本节的其他方法涉及增大和收缩向量。数组和切片的大小是不可变的，因此没有这些方法。

向量的所有元素都存储在分配在堆的连续内存块中。向量的**容量**是这个块中可以容纳的最大元素数量。Vec 通常自动管理容量，在需要更多容量时会分配更大的缓冲区并把元素转移过去。不过也有一些显式管理容量的方法。

- **Vec::with_capacity(n)** 创建容量为 n 的新的空向量。
- **vec.capacity()** 返回 vec 的容量,类型为 usize。vec.capacity() >= vec.len() 是肯定的。
- **vec.reserve(n)** 确保向量有足够空间容纳另外 n 个元素。换句话说，vec.capacity() 至少等于 vec.len() + n。如果容量已经够了，则什么也不做。否则，就会分配一个更大的缓冲区，然后把向量内容转移过去。
- **vec.reserve_exact(n)** 与 vec.reserve(n) 类似，但告诉 vec 分配的空间不能超过 n，不考虑未来的增长。之后，vec.capacity() 等于 vec.len() + n。
- 如果 vec.capacity() 大于 vec.len()，**vec.shrink_to_fit()** 则尝试释放额外的内存。

Vec<T> 有很多添加和移除元素的方法，它们可以改变向量的长度。这些方法都以向量的可修改（mut）self 引用为参数。

下面两个方法在向量末尾添加或移除一个值。

- **vec.push(value)** 在 vec 末尾添加给定的 value。

- **vec.pop()** 移除并返回最后一个元素。返回值的类型是 Option<T>。如果最后一个元素是 x 则返回 Some(x)，如果向量为空则返回 None。

注意，.push() 取得参数的值，而不是引用。类似地，.pop() 也返回元素的值而非引用。本节剩下的多数方法也是如此。它们可以给向量添加值或者从向量中移除值。

下面两个方法在向量的任何地方添加或移除值。

- **vec.insert(index, value)** 在 vec[index] 中插入 value，将 vec[index..] 中的值向右顺移一个位置。
  如果 index > vec.len() 则诧异。
- **vec.remove(index)** 移除并返回 vec[index]，将 vec[index+1..] 中的值向左顺移一个位置。
  如果 index >= vec.len() 则诧异，因为此时 vec[index] 没有要移除的元素。
  向量越长，这个操作越慢。如果经常需要 vec.remove(0)，可以使用 VecDeque（参见 16.3 节）而不是 Vec。

要移动的元素越多，.insert() 和 .remove() 的速度越慢。

下面 3 个方法可以将向量的长度修改为指定的值。

- **vec.resize(new_len, value)** 将向量长度设置为 new_len。如果这样 vec 的长度会增长，则将 value 的副本放到新位置。元素的类型必须实现 Clone 特型。
- **vec.truncate(new_len)** 将向量长度减少为 new_len，清除范围在 vec[new_len..] 之内的元素。
  如果 vec.len() 小于或等于 new_len，则什么也不会发生。
- **vec.clear()** 从 vec 中移除所有元素。相当于 vec.truncate(0)。

以下 4 个方法可以一次添加或移除多个值。

- **vec.extend(iterable)** 按顺序将 iterable 的所有项添加到 vec 末尾。类似于给 .push() 传入多个值。iterable 参数可以是实现 IntoIterator<Item=T> 的任何值。

  这个方法非常有用，因此它有一个专门的标准特型 Extend，所有标准集合都实现了。然而，这也导致在 rustdoc 生成的 HTML 中把 .extend() 和其他特型方法都混在了一起，挤到了页面底部，因此在需要的时候很难找到它。只要记住它就在那里就行了。更多信息参见 15.4.13 节。

- **vec.split_off(index)** 与 vec.truncate(index) 类似，只不过它会返回一个包含从 vec 末尾移除值的 Vec<T>。这就类似多值版的 .pop()。
- **vec.append(&mut vec2)** 把 vec2 的所有元素转移到 vec，之后 vec2 变空。vec2 也是类型为 Vec<T> 的向量。

  与 vec.extend(vec2) 类似，只不过 vec2 在操作之后还存在，容量不会受影响。

- **vec.drain(range)** 从 vec 中移除范围 vec[range]，返回被移除元素的迭代器。range 是一个范围值，类似 .. 或 0..4。

还有几个奇怪的方法，可以从向量中选择性地移除元素。

- **vec.retain(test)** 移除所有没有通过给定测试的元素。test 参数是一个实现 FnMut(&T) -> bool 的函数或闭包。对 vec 的每个元素，都会调用 test(&element)，如果返回 false，则从向量中移除 element 并清除。

  不考虑性能，就类似于如下代码：

  ```
  vec = vec.into_iter().filter(test).collect();
  ```

- **vec.dedup()** 清除重复的元素。类似于 Unix 的 uniq 终端命令。这个方法会扫描 vec，对比相邻的元素，如果发现相等则清除额外相等的值，只留下一个：

  ```
  let mut byte_vec = b"Missssssissippi".to_vec();
  byte_vec.dedup();
  assert_eq!(&byte_vec, b"Misisipi");
  ```

  注意，结果中仍然有两个 's'。这个方法只移除**连续的**重复值。要去掉所有重复的值，有 3 个办法。一是调用 .dedup() 之前先把向量排序，二是把数据转移到集中，三是使用（可以保持元素原始顺序的）.retain() 技巧：

  ```
  let mut byte_vec = b"Missssssissippi".to_vec();

  let mut seen = HashSet::new();
  byte_vec.retain(|r| seen.insert(*r));

  assert_eq!(&byte_vec, b"Misp");
  ```

  这之所以可行，是因为 .insert() 在集中已经包含要插入项时返回 false。

- **vec.dedup_by(same)** 与 vec.dedup() 类似，只不过它使用函数或闭包 same(&mut elem1, &mut elem2) 而不是 == 操作符判断两个元素是不是重复。
- **vec.dedup_by_key(key)** 与 vec.dedup() 类似，只不过判断两个元素重复的依据是 key(&mut elem1) == key(&mut elem2)。

  例如，如果 error 是一个 Vec<Box<Error>>，可以这样写：

  ```
  // 移除消息内容重复的错误
  errors.dedup_by_key(|err| err.description().to_string());
  ```

本节介绍的所有方法中只有 .resize() 会克隆值。其他方法都是把值从一个地方转移到另一个地方。

16.2.4 连接

以下两个方法适用于**数组的数组**，包括任何数组、切片或向量，其元素本身也是数组、切片或向量。

- **slices.concat()** 返回拼接所有切片得到的新向量。

  ```
  assert_eq!([[1, 2], [3, 4], [5, 6]].concat(),
             vec![1, 2, 3, 4, 5, 6]);
  ```

- **slices.join(&separator)** 与 slices.concat() 类似，只不过会把 separator 的副本插到切片之间：

```
assert_eq!([[1, 2], [3, 4], [5, 6]].join(&0),
           vec![1, 2, 0, 3, 4, 0, 5, 6]);
```

16.2.5　拆分

同时取得多个不可修改引用的数组、切片或向量很容易：

```
let v = vec![0, 1, 2, 3];
let a = &v[i];
let b = &v[j];

let mid = v.len() / 2;
let front_half = &v[..mid];
let back_half = &v[mid..];
```

但取得多个可修改引用就不容易了：

```
let mut v = vec![0, 1, 2, 3];
let a = &mut v[i];
let b = &mut v[j]; // error: cannot borrow `v` as mutable
                   //        more than once at a time
```

Rust 禁止这样操作，因为如果 `i == j`，那么 `a` 和 `b` 就都是对同一个整数的可修改引用，这违反了 Rust 的安全规则。（参见 5.3 节。）

Rust 有可以同时将多个可修改引用借用为两个或多个数组、切片或向量的方法。与上面的代码不同，这些方法是安全的，因为它们从设计上就保证了把数据拆分为**不重叠**的区块，其中有许多方法还适用于不可修改的切片，因此就有了 `mut` 和非 `mut` 两个版本。

图 16-2 展示了这些方法。这些方法都不直接修改数组、切片或向量，它们仅仅返回对其中部分数据的新引用。

- **`slice.iter()`** 和 **`slice.iter_mut()`** 返回的迭代器产生 `slice` 中每个元素的引用。16.2.2 节介绍过它们。
- **`slice.split_at(index)`** 和 **`slice.split_at_mut(index)`** 将切片一分为二，返回包含两个部分的元组。`slice.split_at(index)` 等价于 `(&slice[..index], &slice[index..])`。如果 `index` 越界，这两个方法会诧异。
- **`slice.split_first()`** 和 **`slice.split_first_mut()`** 也返回元组，包含对第一个元素的引用（`slice[0]`）和对所有剩余元素切片的引用（`slice[1..]`）。

  `.split_first()` 返回值的类型是 `Option<(&T, &[T])>`，如果切片为空则返回 `None`。

- **`slice.split_last()`** 和 **`slice.split_last_mut()`** 类似，但以最后一个而不是第一个元素为分隔符。

  `.split_last()` 返回值的类型是 `Option<(&[T], &T)>`。

- **`slice.split(is_sep)`** 和 **`slice.split_mut(is_sep)`** 基于函数或闭包 `is_sep` 将 `slice` 拆分为一个或多个子切片，返回子切片的迭代器。

  在消费返回的迭代器时，它会对切片中的每个元素调用 `is_sep(&element)`。如果

is_sep(&element) 返回 true，则这个元素就是分隔符。分隔符元素不包含在输出的任何子切片中。

输出始终包含至少一个子切片，多一个分隔符元素就多一个子切片。如果多个分隔符相邻或者分隔符是 slice 的最后一个元素，则输出会包含空子切片。

- **slice.splitn(n, is_sep)** 和 **slice.splitn_mut(n, is_sep)** 类似，但最多只产生 *n* 个子切片。在找到前 *n*–1 个切片后，就不会再调用 is_sep。最后一个子切片包含所有剩余元素。
- **slice.rsplitn(n, is_sep)** 和 **slice.rsplitn_mut(n, is_sep)** 与 .splitn() 和 .splitn_mut() 类似，只是反向扫描切片。换句话说，这两个方法会基于切片中的**后**（而不是前）*n*–1 个分隔符拆分，因此会从末尾开始产生子切片。
- **slice.chunks(n)** 和 **slice.chunks_mut(n)** 返回迭代器，产生长度为 *n* 的非重叠子切片。如果 slice.len() 不是 *n* 的倍数，则最后一个子切片长度小于 *n*。

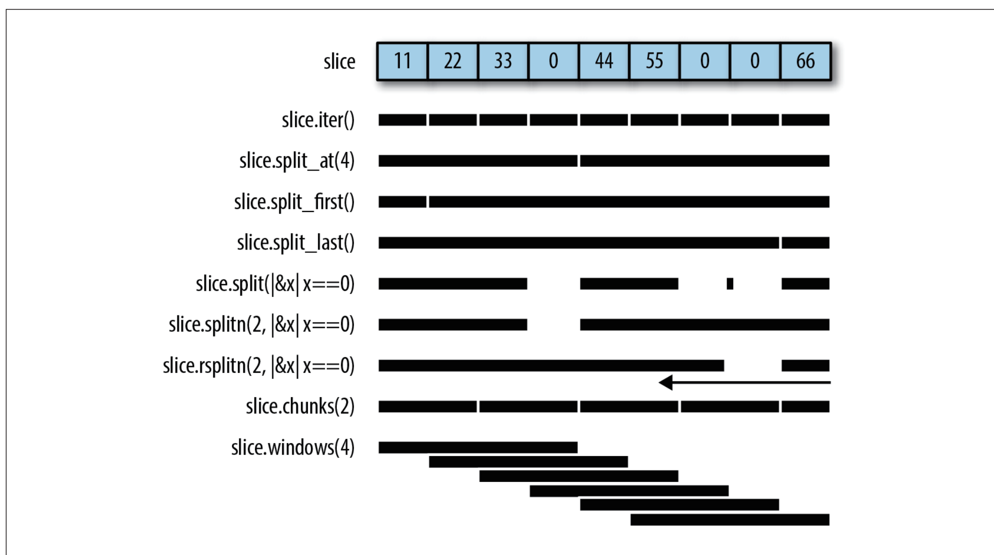


图 16-2：拆分方法示意图。slice.split() 输出中的小矩形是一个空切片，因为两个分隔符相邻。另外，rsplitn 是从后向前产生输出，与其他方法不同

还有一个迭代子切片的方法。

- **slice.windows(n)** 返回的迭代器类似于 slice 数据的“滑动窗口”。这个迭代器产生包含 slice 中 *n* 个连续元素的子切片。比如，产生的第一个子切片是 &slice[0..n]，第二个子切片是 &slice[1..n+1]，以此类推。
  如果 *n* 大于 slice 的长度，则不会产生切片。如果 *n* 是 0，则这个方法会诧异。
  例如，如果 days.len() == 31，那么调用 days.windows(7) 会产生 days 中所有连续 7 天的子切片。

  大小为 2 的滑动窗口可用于检测数据序列中两个数据点的变化：

```
let changes = daily_high_temperatures
                  .windows(2)          // 取得相邻两天的温度
                  .map(|w| w[1] - w[0]) // 温差有多大?
                  .collect::<Vec<_>>();
```

因为子切片是重叠的，所以这个方法没有返回 mut 引用的变体。

16.2.6 交换

交换两个元素也有相应的方法。

- **slice.swap(i, j)** 会交换 slice[i] 和 slice[j] 这两个元素。

向量还支持一个相对方法，此方法可以高效移除任意元素。

- **vec.swap_remove(i)** 移除并返回 vec[i]，跟 vec.remove(i) 类似，但不会把后面的元素向前移动以填补空缺，而是简单地把 vec 的最后一个元素移到空位上。在不关心向量剩余元素顺序的情况下可以使用。

16.2.7 排序和搜索

切片提供了 3 个排序方法。

- **slice.sort()** 会对元素递增排序。这个方法只在元素类型实现 Ord 的情况下才存在。
- **slice.sort_by(cmp)** 使用指定排序方式的函数或闭包 cmp 对 slice 的元素进行排序。cmp 必须实现 Fn(&T, &T) -> std::cmp::Ordering。

  手工实现 cmp 比较麻烦，可以委托给一个 .cmp() 方法：

  ```
  students.sort_by(|a, b| a.last_name.cmp(&b.last_name));
  ```

  如果想按某字段排序，用另一个字段作为最终依据，则要比较元组：

  ```
  students.sort_by(|a, b| {
      let a_key = (&a.last_name, &a.first_name);
      let b_key = (&b.last_name, &b.first_name);
      a_key.cmp(&b_key)
  });
  ```

- **slice.sort_by_key(key)** 会按照排序键对 slice 的元素递增排序，排序键使用函数或闭包 key 给出。key 的类型必须实现 Fn(&T) -> K where K: Ord。

  在 T 包含一个或多个可排序字段时可以使用，可以选择多种排序方式：

  ```
  // 按年级平均分排序，最低分排前头
  students.sort_by_key(|s| s.grade_point_average());
  ```

  注意，排序期间不会缓存排序键的值，因此 key 函数可能会被调用 *n* 次以上。

  由于技术上的原因，key(element) 不能返回从元素借用的引用。这样不行：

  ```
  students.sort_by_key(|s| &s.last_name);  // 错误：不能推断生命期
  ```

  Rust 无法确定生命期。但在这种情况下，直接使用 .sort_by() 也没问题。

以上 3 个方法都执行稳定排序。

要反向排序，可以使用 sort_by 并传入两次参数调换了次序的 cmp 闭包。比如，|b, a| 而不是 |a, b| 就会执行反向排序。或者也可以在排序之后调用 .reverse() 方法。

- **slice.reverse()** 对切片元素就地反向排序。

切片排序完毕后，可以高效地进行搜索。

- **slice.binary_search(&value)**、**slice.binary_search_by(&value, cmp)** 和 **slice.binary_search_by_key(&value, key)** 都可以从排序后的切片中搜索 value。注意 value 传的是引用。

  这几个方法返回的类型是 Result<usize, usize>。如果在指定的排序下 slice[index] 等于 value，它们就返回 Ok(index)。如果没有找到这个索引，它们则返回 Err(insertion_point)，这样在 insertion_point 位置插入 value 会保持顺序。

当然，二分查找只对已经按指定方式排过序的切片有用。否则结果无法确定，即“垃圾进，垃圾出”。

因为 f32 和 f64 有 NaN 值，所以它们都未能实现 Ord，也不能直接作为键用于排序和二分查找方法。要对浮点数据使用类似的方法，可以选择 ord_subset 包。

还有一个搜索未排序向量的方法。

- **slice.contains(&value)** 在 slice 的任意元素等于 value 时返回 true。这个方法逐个检查切片的元素看是否匹配。同样，value 传的是引用。

要查找值在切片中的位置，类似 JavaScript 中的 array.indexOf(value)，可以使用迭代器：

```
slice.iter().position(|x| *x == value)
```

返回的是一个 Option<usize>。

16.2.8 比较切片

如果类型 T 支持 == 和 != 操作符（PartialEq 特型，参见 12.2 节），那么数组 [T; N]、切片 [T] 和向量 Vec<T> 也支持它们。如果切片的长度相等，对应的每个元素也相等，那它们就相等。数组和向量也是如此。

如果 T 支持 <、<=、> 和 >= 操作符（PartialOrd 特型，参见 12.3 节），那么 T 的数组、切片和向量也支持。切片比较是按词典顺序进行的。

以下是两个常用的切片比较方法。

- **slice.starts_with(other)** 在 slice 开头的一系列值等于另一个切片 other 的元素时会返回 true。

```
assert_eq!([1, 2, 3, 4].starts_with(&[1, 2]), true);
assert_eq!([1, 2, 3, 4].starts_with(&[2, 3]), false);
```

- **slice.ends_with(other)** 类似，但比较的是 slice 的末尾。

```
assert_eq!([1, 2, 3, 4].ends_with(&[3, 4]), true);
```

16.2.9　随机元素

随机数并没有内置在 Rust 标准库中。rand 包提供了以下两个方法，用于从数组、切片或向量中取得随机输出。

- **rng.choose(slice)** 返回对 slice 中随机元素的引用。与 slice.first() 和 slice.last() 类似，这个方法也返回 Option<&T>，只有在切片为空时会返回 None。
- **rng.shuffle(slice)** 对 slice 元素就地随机排序。必须传入 slice 的可修改（mut）引用。

这两个是 rand::Rng 特型的方法，因此调用它们需要一个 Rng（随机数生成器）。不过，通过调用 rand::thread_rng() 可以方便地取得它。要打乱 my_vec 的顺序，可以这样：

```
use rand::{Rng, thread_rng};

thread_rng().shuffle(&mut my_vec);
```

16.2.10　Rust排除无效错误

大多数主流编程语言支持集合和迭代器，也会有这条规则：不要在迭代集合的时候修改它。例如，Python 中与向量对应的是列表：

```
my_list = [1, 3, 5, 7, 9]
```

假设要从 my_list 中移除大于 4 的所有值：

```
for index, val in enumerate(my_list):
    if val > 4:
        del my_list[index] # bug：迭代过程中修改列表

print(my_list)
```

（Python 的 enumerate 函数对应 Rust 的 .enumerate() 方法，参见 15.3.11 节。）

这个程序会令人吃惊地打印 [1, 3, 7]；但 7 大于 4。它是怎么漏网的呢？这是因为一个无效错误导致的：程序在迭代期间修改数据导致了迭代器**无效**。在 Java 中，这会导致异常；在 C++ 中，这是一个未定义行为。在 Python 中，虽然这个行为有明确定义，但不直观：迭代器跳过一个元素。因此 val 永远不会等于 7。

下面我们在 Rust 中复现这个 bug：

```
fn main() {
    let mut my_vec = vec![1, 3, 5, 7, 9];
    for (index, &val) in my_vec.iter().enumerate() {
        if val > 4 {
            my_vec.remove(index); // 错误：不能借用my_vec的可修改引用
        }
    }
    println!("{:?}", my_vec);
}
```

自然地，Rust 在编译时就拒绝了这个程序。在调用 `my_vec.iter()` 时，循环会借用向量的共享（不可修改）引用。这个引用的生命期与迭代器一样长，直到 `for` 循环结束。在存在不可修改引用的情况下，不能调用 `my_vec.remove(index)` 修改向量。

编译时报错当然很好，但还需要想办法达到目的。修复这个问题最简单的方式是这样写：

```
my_vec.retain(|&val| val <= 4);
```

或者在 Python 和其他语言中，可以使用 `filter` 创建一个新向量。

16.3 VecDeque<T>

`Vec` 只支持在两端高效添加和移除元素。如果程序需要把值保存在队列中间，`Vec` 就慢了。

Rust 的 `std::collections::VecDeque<T>` 是一个 deque（发音 "deck"），即双端队列。它支持前端和后端的高效添加和移除操作。

- **`deque.push_front(value)`** 在队列前面添加值。
- **`deque.push_back(value)`** 在队列后面添加值。（这个方法用得比 `.push_front()` 多，因为队列习惯上都在后面添加值并在前面移除值，就像人们排队一样。）
- **`deque.pop_front()`** 移除并返回队列前面的值，返回 `Option<T>`，队列为空时返回 `None`，跟 `vec.pop()` 一样。
- **`deque.pop_back()`** 移除并返回队列后面的值，同样返回 `Option<T>`。
- **`deque.front()`** 和 **`deque.back()`** 与 `vec.first()` 和 `vec.last()` 相似，返回队列前端和后端元素的引用。返回 `Option<T>`，队列为空时返回 `None`。
- **`deque.front_mut()`** 和 **`deque.back_mut()`** 与 `vec.first_mut()` 和 `vec.last_mut()` 相似，返回 `Option<&mut T>`。

`VecDeque` 的实现是环形缓冲区，如图 16-3 所示。

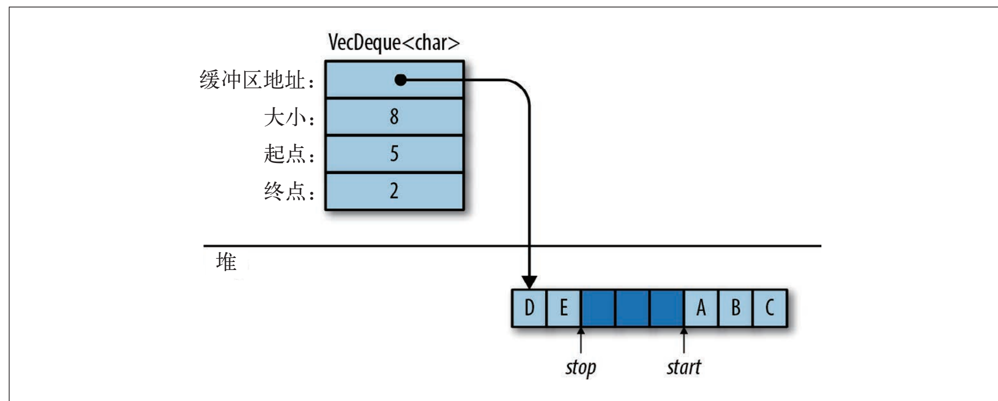


图 16-3：VecDeque 在内存中的布局

跟 Vec 一样，VecDeque 在堆中有一块内存保存元素。但与 Vec 不同，VecDeque 的数据在这块内存中并不总是从头开始存储，是可以环绕回来的，如图所示。按顺序排列，这个队列中的元素为 ['A', 'B', 'C', 'D', 'E']。VecDeque 有私有字段，图中以“起点”和“终点”标识，用于保存数据在缓冲区的开始和结束位置。

向队列中任何一端添加值都意味着需要一个未被使用的槽位，即图中的深灰色区块。添加的值与两端接触，如果空间不够则分配更多内存。

VecDeque 自动管理环绕，无须开发者操心。图 16-3 是 Rust 之所以让 .pop_front() 很快的底层视图。

一般来说在使用双向队列时，只需要使用 .push_back() 和 .pop_front() 两个方法。静态方法 VecDeque::new() 和 VecDeque::with_capacity(n) 可以用来创建队列，与对应的 Vec 方法一样。Vec 的很多方法也在 VecDeque 上实现了，比如 .len() 和 .is_empty()、.insert(index, value) 和 .remove(index)、.extend(iterable)，等等。

与向量一样，队列可以按值、按共享引用或按可修改引用迭代。它们有 3 个迭代器方法：.into_iter()、.iter() 和 .iter_mut()。可以按惯常的方式通过索引来访问，比如 deque[index]。

不过，由于队列的值在内存中不是连续存储的，因此它们并没有继承切片的所有方法。一种在队列数据上执行向量和切片操作的方式是把 VecDeque 转换为 Vec，执行操作，然后再转换回去。

- Vec<T> 实现了 From<VecDeque<T>>，因此 **Vec::from(deque)** 可以将队列转换为向量。这个操作的时间复杂度为 $O(n)$，因为可能需要重排元素。
- VecDeque<T> 实现了 From<Vec<T>>，因此 **VecDeque::from(vec)** 可以将向量转换为队列。这个操作的时间复杂度也是 $O(n)$，但通常会更快一些，即便向量很大。原因在于向量的堆内存可以直接转移给新队列。

  使用这个方法可以方便地通过指定元素来创建队列，尽管没有标准的 vec_deque![] 宏：

  ```
  use std::collections::VecDeque;

  let v = VecDeque::from(vec![1, 2, 3, 4]);
  ```

16.4 LinkedList<T>

链表是另一种存储序列值的方式。链表的每个值都存储在独立的堆内存中，如图 16-4 所示。

std::collections::LinkedList<T> 是 Rust 的双向链表，支持 VecDeque 的部分方法，包括操作序列前后端的方法、迭代器方法、LinkedList::new() 和其他几个方法。不过，通过索引访问元素的方法通常就没有了，因为通过索引访问链表元素效率本身就很低。

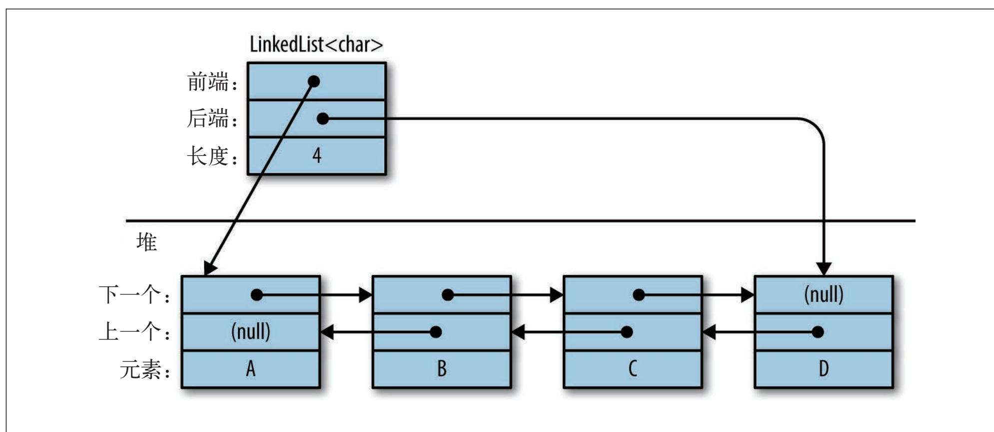


图 16-4：LinkedList<char> 的内存布局

到 Rust 1.17 为止，Rust 的 LinkedList 类型还没有从列表中移除某个范围内元素的方法，也没有在列表中特定位置插入元素的方法。当然，其 API 还在不断完善。

目前来看，LinkedList 比 VecDeque 强大的地方主要体现在组合两个列表的速度非常快。list.append(&mut list2) 会把 list2 的元素都转移到 list 中，只需修改几个指针即可，因此时间复杂度是常量。Vec 和 VecDeque 的 append 方法有时候必须把很多值从一个堆阵列转移到另一个。

16.5 BinaryHeap<T>

BinaryHeap 是一个元素松散组织的集合，而且最大值始终会冒泡到队列前端。以下是 3 个最常用的 BinaryHeap 方法。

- **heap.push(value)** 向堆中添加值。
- **heap.pop()** 从堆中移除并返回最大值。返回类型为 Option<T>，如果堆为空则返回 None。
- **heap.peek()** 返回堆中最大值的引用。返回类型为 Option<&T>。

BinaryHeap 也支持 Vec 的部分方法，包括 BinaryHeap::new()、.len()、.is_empty()、.capacity()、.clear() 和 .append(&mut heap2)。

如果像下面这样给 BinaryHeap 填充一批数值：

```
use std::collections::BinaryHeap;

let mut heap = BinaryHeap::from(vec![2, 3, 8, 6, 9, 5, 4]);
```

那么数值 9 就会跑到堆的顶部：

```
assert_eq!(heap.peek(), Some(&9));
assert_eq!(heap.pop(), Some(9));
```

移除数值 9 还会导致其他值重排，之后数值 8 会跑到顶部，以此类推：

```
assert_eq!(heap.pop(), Some(8));
assert_eq!(heap.pop(), Some(6));
assert_eq!(heap.pop(), Some(5));
...
```

当然，BinaryHeap 并不限于存储数值。实现内置特型 Ord 的任何类型都可以保存在 BinaryHeap 中。

BinaryHeap 可以用作工作队列。比如，定义一个任务结构体，基于优先级实现 Ord，让高优先级任务大于低优先级任务。然后，创建一个 BinaryHeap 保存所有待完成的任务。调用 .pop() 方法始终返回接下来要做的最重要的任务。

注意，BinaryHeap 是可迭代类型，也有自己的 .iter() 方法，但迭代器会以任意顺序产生堆的元素，而不是从大到小。要按优先级顺序消费 BinaryHeap 中的值，需要使用 while 循环：

```
while let Some(task) = heap.pop() {
    handle(task);
}
```

16.6 HashMap<K, V>和BTreeMap<K, V>

映射（map）是一种键–值对（称作**条目**）集合。条目的键唯一，而且条目的组织保证可以通过键高效获取对应的值。简单来说，映射就是一个查找表。

Rust 提供两种映射类型：HashMap<K, V> 和 BTreeMap<K, V>。这两种类型共享了很多方法，区别在于两者为快速查找而设计的条目布局。

HashMap 把键和值保存在一个散列表中，因此要求键的类型 K 实现标准的散列和相等性特型 Hash 和 Eq。

图 16-5 展示了 HashMap 在内存中的布局。空白区块是未使用的。所有键、值和缓存的散列码都存储在一个分配在堆上的表中。添加条目会强制 HashMap 分配更大的表，然后把所有数据都转移进去。

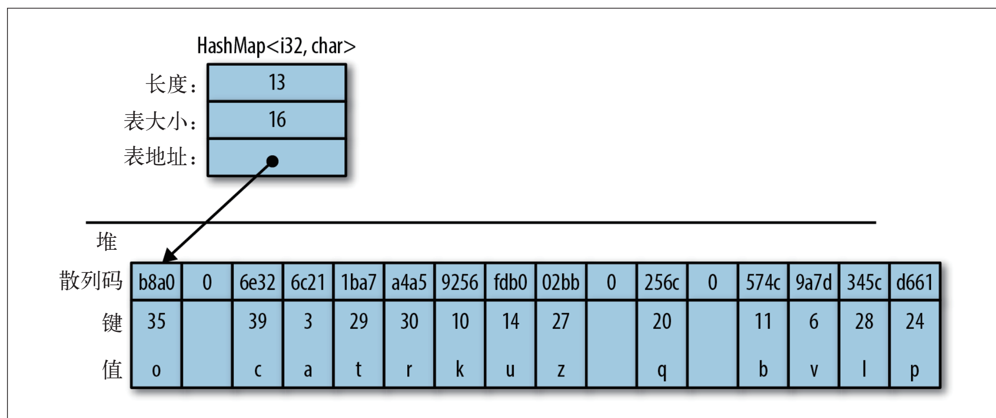


图 16-5：HashMap 的内存布局

BTreeMap 按照键的顺序存储条目，总体为树形结构，因此要求键类型 K 实现 Ord。图 16-6 展示了 BTreeMap 的存储方式。深色区块表示未使用闲置容量。

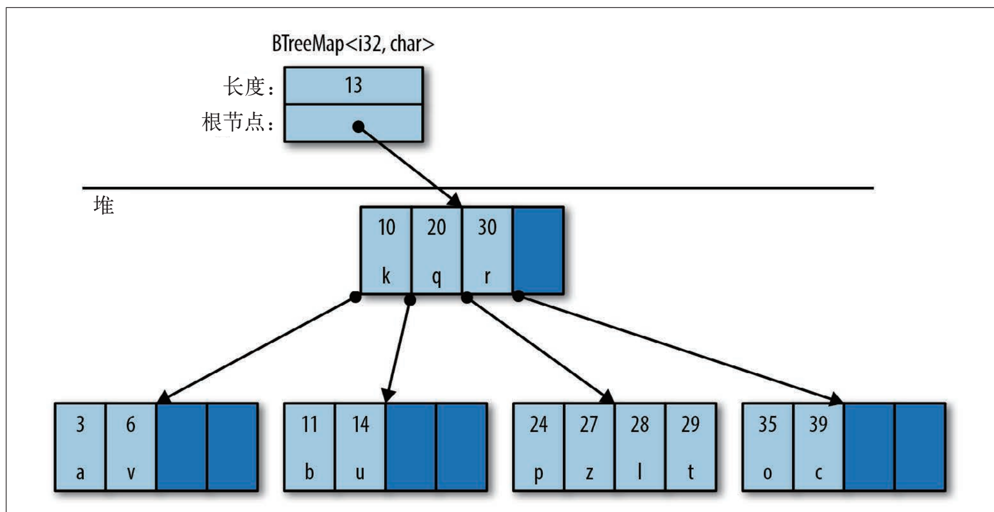


图 16-6：BTreeMap 的内存布局

BTreeMap 将条目存储在**节点**中，其中大多数节点只包含键 – 值对。非叶节点，即图中的根节点，也有保存子节点指针的空间。(20, 'q') 和 (30, 'r') 之间的指针指向的子节点包含的键介于 20 到 30 之间。添加条目经常需要将已有节点的条目向右移动以保持顺序，偶尔也需要分配新节点。

图 16-6 为适应页面大小进行了简化。真正的 BTreeMap 节点应该有 11 个位置，而不止 4 个。

Rust 标准库使用 B 树而不使用平衡二叉树的原因是 B 树在现代硬件上更快。二叉树每次搜索所需比较的次数可能比 B 树更少，但搜索 B 树的**领域性**（locality）更好。换句话说，内存访问集中在一块，而不是分散到整个堆上。这样 CPU 缓存就很少错失，从而显著提升速度。

可以使用如下方法创建映射。

- **HashMap::new()** 和 **BTreeMap::new()** 创建新的空映射。
- **iter.collect()** 可用于从键 – 值对创建和填充新 HashMap 或 BTreeMap。iter 必须是一个 Iterator<Item=(K, V)>。
- **HashMap::with_capacity(n)** 创建新的空散列映射，至少能容纳 *n* 个条目。与向量一样，HashMap 也是将数据存储在一块堆内存中，因此就有容量相关的方法 hash_map.capacity()、hash_map.reserve(additional) 和 hash_map.shrink_to_fit()。BTreeMap 则没有这些方法。

HashMap 和 BTreeMap 拥有与键和值相关的相同的核心方法。

- **map.len()** 返回条目数量。

- **map.is_empty()** 在 map 没有条目时返回 true。
- **map.contains_key(&key)** 在映射有条目的键为 key 时返回 true。
- **map.get(&key)** 搜索 map 中具有给定 key 的条目。如果找到了匹配的条目，就返回 Some(r)，其中 r 是对相应值的引用。否则返回 None。
- **map.get_mut(&key)** 类似，但返回对值的可修改引用。

  一般来说，映射允许对其存储值的可修改访问，但不允许修改键。对于值可以随意修改，但键属于映射本身，必须确保不改，因为条目就是按照键来组织的。直接修改键属于 bug。

- **map.insert(key, value)** 向 map 中插入条目 (key, value)。如果映射中已经存在键为 key 的条目，则新插入的 value 会覆盖原来的值。

  如果覆盖了原来的值，则返回原来的值。返回类型为 Option<V>。

- **map.extend(iterable)** 迭代 iterable 的 (K, V) 条目，并将每个键 – 值对插入 map 中。
- **map.append(&mut map2)** 将 map2 的所有条目转移到 map 中。之后，map2 变空。
- **map.remove(&key)** 查找并移除 map 中键为 key 的条目。
  如果有值被移除，则返回移除的值。返回类型为 Option<V>。
- **map.clear()** 移除所有条目。

还可以使用方括号语法 **map[&key]** 查询映射。这意味着映射实现了 Index 内置特型。不过，如果不存在键为 key 的条目则会诧异，比如数组访问越界。因此要保证想查找的条目确实存在时再使用这种语法。

.contains_key()、.get()、.get_mut() 和 .remove() 的 key 参数不一定正好是类型 &K。这几个方法对于可以从 K 借用的类型是通用的。比如，在 HashMap<String, Fish> 上调用 fish_map.contains_key("conger") 是没问题的，虽然 "conger" 并不是 String 类型，但 String 实现了 Borrow<&str>。要了解详细信息，参见 13.8 节。

因为 BTreeMap<K, V> 按键来排序并存储条目，所以它还支持下面这个操作。

- **btree_map.split_off(&key)** 将 btree_map 一分为二。键小于 key 的条目会留在 btree_map 中。而返回的新 BTreeMap<K, V> 包含其他条目。

16.6.1　条目

HashMap 和 BTreeMap 都有对应的 Entry 类型。Entry（条目）类型的用意在于消除多余的映射查找。例如，下面的代码会取得或创建一条学生记录：

```
// 已经有这个学生的记录了吗?
if !student_map.contains_key(name) {
    // 没有：创建一个
    student_map.insert(name.to_string(), Student::new());
}
// 现在绝对有这个学生的记录了
let record = student_map.get_mut(name).unwrap();
...
```

这样当然没问题，但会访问两次或 3 次 student_map，每次都执行同样的查找操作。

而使用条目的原因在于可以只查找一次，然后产生一个 Entry 值供后续操作使用。下面这一行代码等价于前面的所有代码，但只有一次查找：

```
let record = student_map.entry(name.to_string()).or_insert_with(Student::new);
```

student_map.entry(name.to_string()) 返回的 Entry 值类似于指向映射中某个位置的可修改引用，要么已经被一个键 – 值对**占用**，要么是**空着的**（vacant），意味着那里还没有条目。如果是空着的，则条目的 .or_insert_with() 方法会插入一个新 Student。条目的多数用法与此类似，简单明了。

同样的方法也可以创建 Entry 值。

- **map.entry(key)** 返回键为 key 的 Entry。如果映射中没有这个 key，则返回一个空 Entry。

  这个方法接收自己的可修改引用，返回生命期匹配的 Entry：

  ```
  pub fn entry<'a>(&'a mut self, key: K) -> Entry<'a, K, V>
  ```

  这里的 Entry 类型有一个生命期参数 'a，这是因为它实际上只是对映射的一个借用 mut 引用。只要 Entry 存在，它就拥有对映射的专有访问权。

  5.2.5 节介绍了如何在一个类型中存储引用，以及这样做对生命期的影响。现在我们从用户角度看到了这种情况。Entry 就是这样的。

  可惜的是，如果映射的键是 String 类型的，则不能给这个方法传 &str 类型的引用。此时，要给 .entry() 方法传入真正的 String。

Entry 值提供了两个填充空条目的方法。

- **map.entry(key).or_insert(value)** 保证 map 包含键为 key 的条目，必要时会以给定的默认 value 插入新条目。这个方法返回对新的或已有值的可修改引用。
  假设我们要计算投票数。可以这样写：

  ```
  let mut vote_counts: HashMap<String, usize> = HashMap::new();
  for name in ballots {
      let count = vote_counts.entry(name).or_insert(0);
      *count += 1;
  }
  ```

  .or_insert() 返回可修改的引用，因此这里 count 的类型是 &mut usize。

- **map.entry(key).or_insert_with(default_fn)** 也一样，只不过在需要创建新条目时会调用 default_fn() 产生默认值。如果 map 中已经存在键为 key 的条目，则不会用到 default_fn。
  假设我们想知道哪些词出现在了哪些文件中。可以这样写：

  ```
  // 这个映射包含每个单词出现过的所有文件
  let mut word_occurrence: HashMap<String, HashSet<String>> =
      HashMap::new();
  for file in files {
  ```

```
        for word in read_words(file)? {
            let set = word_occurrence
                .entry(word)
                .or_insert_with(HashSet::new);
            set.insert(file.clone());
        }
    }
```

Entry 是一个枚举类型，HashMap 的 Entry 定义类似下面这样（BTreeMap 的 Entry 定义也类似）：

```
// （在std::collections::hash_map中）
pub enum Entry<'a, K: 'a, V: 'a> {
    Occupied(OccupiedEntry<'a, K, V>),
    Vacant(VacantEntry<'a, K, V>)
}
```

这里的 OccupiedEntry 和 VacantEntry 类型都有插入、移除和访问条目而不重复查找的方法。可以通过在线文档来查阅这些方法。偶尔，通过这些方法可以消除一两次多余的查找，但 .or_insert() 和 .or_insert_with() 已经涵盖了常见情况。

16.6.2　映射迭代

迭代映射有以下几种方式。

- 按值迭代（for (k, v) in map）产生 (K, V) 对。这样会消费映射。
- 按共享引用迭代（for (k, v) in &map）产生 (&K, &V) 对。
- 按可修改引用迭代（for (k, v) in &mut map）产生 (&K, &mut V) 对。（同样，无法对映射中的键进行可修改访问，因为条目就是按照自己的键来组织的。）

就像向量一样，映射也有返回迭代器且产生值引用的 .iter() 和 .iter_mut() 方法，分别与迭代 &map 和 &mut map 类似。此外，还有如下迭代方法。

- **map.keys()** 返回产生键引用的迭代器。
- **map.values()** 返回产生值引用的迭代器。
- **map.values_mut()** 返回产生可修改值引用的迭代器。

所有 HashMap 迭代器都以任意顺序访问映射的条目。BTreeMap 迭代器按键顺序访问条目。

16.7　HashSet<T>和BTreeSet<T>

集是为快速测试成员关系而组织值的一种集合。

```
let b1 = large_vector.contains("needle");   // 慢，检查每个元素
let b2 = large_hash_set.contains("needle"); // 快，散列查找
```

集中永远不会包含同一个值的多个副本。

映射和集有不同的方法，但在后台集就类似于只有键（而不是键 – 值对）的映射。事实上，Rust 的两个集类型 HashSet<T> 和 BTreeSet<T> 都是以 HashMap<T, ()> 和 BTreeMap<T,

()> 的包装类型形式实现的。

- **HashSet::new()** 和 **BTreeSet::new()** 创建新集。
- **iter.collect()** 可用于从任何迭代器创建新集。如果 iter 产生重复的值，则重复的值会被清除。
- **HashSet::with_capacity(n)** 创建空的 HashSet，至少可以容纳 *n* 个值。

HashSet<T> 和 BTreeSet<T> 有几个共有的简单方法。

- **set.len()** 返回 set 中值的数量。
- **set.is_empty()** 在集中不包含元素时返回 true。
- **set.contains(&value)** 在集中包含给定的 value 时返回 true。
- **set.insert(value)** 向集中添加 value。如果添加成功就返回 true，如果集中已经存在同样的值则返回 false。
- **set.remove(&value)** 从集中移除 value。如果移除成功就返回 true，如果集中已经没有这个值了则返回 false。

与映射类似，按引用查询值的方法也是通用的，只要可以从 T 借用到该类型。要了解详细信息，参见 13.8 节。

16.7.1 集迭代

有两种方法迭代集。

- 按值迭代（for v in set）产生集的成员（并且消费集）。
- 按共享引用迭代（for v in &set）产生集成员的共享引用。

不支持按可修改引用迭代集。没有办法从集中取得值的可修改引用。

- **set.iter()** 返回迭代器，产生 set 成员的引用。

与 HashMap 迭代器一样，HashSet 迭代器也以任意顺序产生值。BTreeSet 迭代器则按顺序产生值，与排序后的向量一样。

16.7.2 相等的值不相同

集有几个奇怪的方法，这些方法只有在关心“相等”的值之间的差异时才会用到。

这种差异确实经常存在。例如，两个相等的 String 值把它们的字符存储在内存中的不同位置：

```
let s1 = "hello".to_string();
let s2 = "hello".to_string();
println!("{:p}", &s1 as &str); // 0x7f8b32060008
println!("{:p}", &s2 as &str); // 0x7f8b32060010
```

通常我们不会在意这种差异。

但万一需要区分，可以使用下列方法取得存储在集中的实际值。这些方法都返回 Option，如果 set 不包含匹配的值则返回 None。

- **set.get(&value)** 返回 set 中等于 value 的成员的共享引用（如果有的话），返回类型是 Option<&T>。
- **set.take(&value)** 类似于 set.remove(&value)，但返回移除的值（如果有的话），返回类型是 Option<T>。
- **set.replace(value)** 类似于 set.insert(value)，但如果 set 已经包含等于 value 的值，那这个方法就返回原来的值。返回类型是 Option<T>。

16.7.3 整集操作

目前，我们看到的大多数集方法只涉及一个集中的一个值。集也有针对整个集的方法。

- **set1.intersection(&set2)** 返回同时包含在 set1 和 set2 中的所有值的迭代器。

  例如，要打印既选了脑外科又选了火箭科学课的学生的名字，可以这样写：

  ```
  for student in brain_class {
      if rocket_class.contains(&student) {
          println!("{}", student);
      }
  }
  ```

  或者更短一点：

  ```
  for student in brain_class.intersection(&rocket_class) {
      println!("{}", student);
  }
  ```

  让人吃惊的是，居然还有一个专门的操作符。

  &set1 & &set2 返回 set1 和 set2 的交集。这是一个二进制按位与操作符，应用到了两个引用。这样就找到了同时在 set1 **和** set2 中的值。

  ```
  let overachievers = &brain_class & &rocket_class;
  ```

- **set1.union(&set2)** 返回同时包含在 set1 和 set2 中，或者只在 set1 或 set2 中的值的迭代器。

  &set1 | &set2 返回包含这两个集中所有值的新集，即查找包含在 set1 **或** set2 中的值。
- **set1.difference(&set2)** 返回包含在 set1 中但不包含在 set2 中的值的迭代器。

  &set1 - &set2 返回包含所有这些值的新集。
- **set1.symmetric_difference(&set2)** 返回只包含在 set1 中或只包含在 set2 中，但不同时包含在两个集中的值的迭代器。

  &set1 ^ &set2 返回包含所有这些值的新集。

下面是测试集与集之间关系的 3 种方法。

- **set1.is_disjoint(set2)** 在 set1 和 set2 没有共同值（即它们的交集为空）时返回 true。
- **set1.is_subset(set2)** 在 set1 是 set2 的子集（即 set1 的所有值也都在 set2 中）时返回 true。
- **set1.is_superset(set2)** 正好相反，在 set1 是 set2 的超集时返回 true。

集也支持 == 和 != 的相等性测试。两个集在包含相同的值时相等。

16.8　散列

std::hash::Hash 是标准库中可散列类型的特型。HashMap 的键和 HashSet 的值必须同时实现 Hash 和 Eq。

大多数实现 Eq 的内置类型也实现了 Hash。整数类型、char 和 String 都是可以散列的，因此只要元组、数组、切片和向量的元素是可以散列的，它们就是可以散列的。

标准库的一个原则是一个值无论保存在哪里或者怎么指向它，都应该有相同的散列码。因此，引用与它指向的值有相同的散列码，而 Box 与装箱的值有相同的散列码。向量 vec 与包含所有其数据的切片 &ved[..] 有相同的散列码。String 与引用相同字符的 &str 有相同的散列码。

结构体和枚举默认没有实现 Hash，但可以派生一个实现：

```
/// 大英博物馆中一件藏品的ID编号
#[derive(Clone, PartialEq, Eq, Hash)]
enum MuseumNumber {
    ...
}
```

只要类型的字段都是可散列的就没问题。

如果手工为一个类型实现了 PartialEq，那么也应该手工为它实现 Hash。例如，假设有一个表示无价历史珍宝的类型：

```
struct Artifact {
    id: MuseumNumber,
    name: String,
    cultures: Vec<Culture>,
    date: RoughTime,
    ...
}
```

如果两个 Artifact 有相同的 ID，那么就认为它们是同一件东西：

```
impl PartialEq for Artifact {
    fn eq(&self, other: &Artifact) -> bool {
        self.id == other.id
    }
}

impl Eq for Artifact {}
```

因为只根据 ID 比较藏品，所以必须使用相同的方式计算它们的散列码：

```
impl Hash for Artifact {
    fn hash<H: Hasher>(&self, hasher: &mut H) {
        // 委托散列到MuseumNumber
        self.id.hash(hasher);
    }
}
```

（否则，HashSet<Artifact> 将不能正确使用。与所有散列表一样，它要求如果 a == b，则必须 hash(a) == hash(b)。）

这样就可以创建 Artifact 的 HashSet 了：

```
let mut collection = HashSet::<Artifact>::new();
```

如代码所示，即使手工实现 Hash，也不需要知道任何有关散列算法的事。.hash() 接收一个 Hasher 的引用，表示散列算法。只要把涉及 == 比较的所有数据传给这个 Hasher 即可。Hasher 会根据传入的数据来计算一个散列码。

使用自定义散列算法

hash 方法是泛型的，因此上面展示的 Hash 实现可以把数据传给实现 Hasher 的任何类型。这也是 Rust 支持可挺拔散列算法的方式。Hash 和 Hasher 是伴型特型，详情参见 11.4.3 节。

第三个特型 std::hash::BuildHasher 针对的是表示散列算法初始状态的类型。每个 Hasher 都是一次性的，类似迭代器，只用一次就不要了。而 BuildHasher 可以重用。

每个 HashMap 都包含一个 BuildHasher，每次计算散列码时都要用到。BuildHasher 值包含键、初始状态或散列算法每次运行时需要的其他参数。

计算散列码的完整协议如下：

```
use std::hash::{Hash, Hasher, BuildHasher};

fn compute_hash<B, T>(builder: &B, value: &T) -> u64
    where B: BuildHasher, T: Hash
{
    let mut hasher = builder.build_hasher(); // 1. 算法开始
    value.hash(&mut hasher);                 // 2. 传入数据
    hasher.finish()                          // 3. 完成，产生u64值
}
```

每次需要计算散列码时，HashMap 都会调用这 3 个方法。所有方法都已行内化，因此非常快。

Rust 默认的散列算法是广为人知的 SipHash-1-3 算法。SipHash 很快，而且散列冲突控制得非常出色。事实上，它是一个加密算法，还没有已知的高效方式可以生成 SipHash-1-3 冲突。只要有不同之处，每个散列表就会使用无法预测的键。因此 Rust 可以防止名为 HashDoS 的拒绝服务攻击，即攻击者故意使用散列冲突来触发服务器中最差的性能。

不过，也许你的应用并不需要防范这种攻击。如果保存很多小键，如整数或非常短的字符串，则有可能实现更快的散列函数。代价就是可能有被 HashDoS 攻击的风险。fnv 包实现了这样一个算法，即 Fowler-Noll-Vo 散列。如果想试一试，可以在 Cargo.toml 中添加这么一行：

```
[dependencies]
fnv = "1.0"
```

然后从 fnv 中导入映射和散列类型：

```
extern crate fnv;

use fnv::{FnvHashMap, FnvHashSet};
```

可以使用这两种类型直接代替 HashMap 和 HashSet。以下是 fnv 的源代码中对上面两个类型的实现：

```
/// 默认使用FNV散列函数的HashMap
pub type FnvHashMap<K, V> = HashMap<K, V, FnvBuildHasher>;

/// 默认使用FNV散列函数的HashSet
pub type FnvHashSet<T> = HashSet<T, FnvBuildHasher>;
```

标准的 HashMap 和 HashSet 集合接收可选的额外类型参数，用于指定散列算法。FnvHashMap 和 FnvHashSet 是 HashMap 和 HashSet 的泛型别名，给这个参数传入了 FNV 散列函数。

16.9　标准集合之外

在 Rust 中创建新的自定义集合类型与在其他语言中大致相同，都需要基于语言提供的构建块（如结构体和枚举、标准集合、Option、Box，等等）来组织数据。参见 10.1.4 节中 BinaryTree<T> 类型的例子。

如果你习惯了在 C++ 中实现数据结构，使用原始指针、手工内存管理、放置 new、显式调用析构函数来获得可能最好的性能，那么无疑会发现安全的 Rust 相当局限。所有这些工具本质上都是不安全的。Rust 也支持这些操作，只是必须选择在不安全的代码中使用。第 21 章会详细介绍如何使用不安全的特性，包括一个使用某些不安全代码实现安全自定义集合的例子。

现在，我们还要继续沐浴在标准集合及其安全、高效 API 温暖的光芒之中。与 Rust 标准库提供的多数特性一样，标准集合的设计初衷就是让编写 unsafe 的代码几乎没有可能。

第17章

字符串与文本

> 字符串是一个赤裸裸的数据结构，其所到之处会有很多重复，因此是隐藏信息的绝佳工具。
>
> ——Alan Perlis，警句 #34

本书已经用到了 Rust 的几种主要文本类型：`String`、`str` 和 `char`。3.5 节中曾介绍过字符和字符串字面量的语法，也展示了字符串在内存中的表示形式。本章将更深入地介绍文本处理。

本章内容如下。

- 有助于理解标准库设计的 Unicode 相关背景知识。
- 介绍表示一个 Unicode 码点的 `char` 类型。
- 介绍 `String` 和 `str` 类型，二者分别表示所有的和借用的 Unicode 字符序列。这两种类型拥有很多方法用于构建、搜索、修改、迭代其内容。
- 介绍 Rust 的字符串格式化实用工具，比如 `println!` 和 `format!` 宏。你也可以编写自定义的宏来处理字符串格式化，或者扩展已有的宏以支持自定义类型。
- 介绍 Rust 对正则表达式的支持。
- 最后谈一谈为什么 Unicode 规范化很重要，以及在 Rust 中如何去做。

17.1 Unicode背景知识

本书是讲 Rust 的，不是讲 Unicode 的。Unicode 已经出了不少书了。但是，Rust 的字符和字符串类型是基于 Unicode 设计的。在此还是有必要介绍一些 Unicode 背景知识，以便更好地理解 Rust。

17.1.1 ASCII、Latin-1和Unicode

Unicode 和 ASCII 中所有的 ASCII 码点是一致的，从 `0` 到 `0x7f` 都一致。比如，二者中字符 `'*'` 的码点都是 42。类似地，Unicode 也将 `0` 到 `0x7f` 分配给了 ISO/IEC 8859-1 字符集中相同的字符，后者是 ASCII 的 8 位超集，用于西方欧洲语言。Unicode 将这个码点范围称为 **Latin-1 编码块**（the Latin-1 code block），因此可以用更好记的名字，即 Latin-1 来称呼 ISO/IEC 8859-1。

由于 Unicode 是 Latin-1 的超集，因此将 Latin-1 转换为 Unicode 甚至连表都不需要：

```
fn latin1_to_char(latin1: u8) -> char {
    latin1 as char
}
```

反向转换也很简单，假设码点都在 Latin-1 的范围内：

```
fn char_to_latin1(c: char) -> Option<u8> {
    if c as u32 <= 0xff {
        Some(c as u8)
    } else {
        None
    }
}
```

17.1.2 UTF-8

Rust 的 `String` 和 `str` 类型使用 UTF-8 编码格式表示文本。UTF-8 将字符编码为 1 到 4 个字节序列，如图 17-1 所示。

UTF-8编码（1~4字节）	码点表示	范围
0xxxxxxx	0bxxxxxxx	0 to 0x7f
110xxxxx 10yyyyyy	0bxxxxxyyyyyy	0x80 to 0x7ff
1110xxxx 10yyyyyy 10zzzzzz	0bxxxxyyyyyyzzzzzz	0x800 to 0xffff
11110xxx 10yyyyyy 10zzzzzz 10wwwwww	0bxxxyyyyyyzzzzzzwwwwww	0x10000 to 0x10ffff

图 17-1：UTF-8 编码

格式良好的 UTF-8 序列有两个限制。第一，对于给定的码点只有最短的编码才被认为是格式良好的，即不能用 4 个字节去编码只需 3 个字节的码点。这条规则确保一个码点只有一个 UTF-8 编码。第二，格式良好的 UTF-8 必须对 `0xd800` 到 `0xdfff`，以及大于 `0x10ffff` 的数值不编码。这些数值要么不用来表示字符，要么完全超出了 Unicode 的范围。

图 17-2 给出了一些例子。

UTF-8编码（1~4字节）	码点表示	范围
00101010	0b0101010 == 0x2a	'*'
11001110 10111100	0b01110_111100 == 0x3bc	'μ'
11101001 10001100 10000110	0b1001_001100_000110 == 0x9306	'寂' (sabi: rust)
11110000 10011111 10100110 10000000	0b000_011111_100110_000000 == 0x1f980	'🦀' (crab emoji)

图 17-2：UTF-8 示例

注意，即便螃蟹的 Emoji 编码的首字节只包含 0，仍然需要 4 个字节来对其编码，因为 3 字节的 UTF-8 编码只能表示 16 位码点，而 `0x1f980` 是 17 位长的。

下面这个简例中的字符串包含编码长度不一的字符：

```
assert_eq!("うどん: udon".as_bytes(),
           &[0xe3, 0x81, 0x86, // う
             0xe3, 0x81, 0xa9, // ど
             0xe3, 0x82, 0x93, // ん
             0x3a, 0x20, 0x75, 0x64, 0x6f, 0x6e // : udon
           ]);
```

图 17-2 展示了一些非常有用的 UTF-8 属性。

- 由于 UTF-8 对码点 `0` 到 `0x7f` 的编码就是字节 `0` 到 `0x7f`，因此保存 ASCII 文本的字节是有效的 UTF-8。如果一个 UTF-8 字符串只包含 ASCII 字符，则反过来说也是正确的：UTF-8 编码是有效的 ASCII。
  Latin-1 与 UTF-8 并不具备这种互逆性。比如，Latin-1 编码 `'é'` 为字节 `0xe9`，UTF-8 会将其解释为一个三字节编码的首字节。
- 通过观察任意字节的前几位，立即就能知道它是某些字符 UTF-8 编码的首字节，还是中间字节。
- 通过编码首字节的前几位就能知道编码的总长度。
- 因为编码最长为 4 个字节，所以 UTF-8 处理不需要无限循环，这对于处理不受信数据非常重要。
- 在格式良好的 UTF-8 中，即便从字节中间的一个随机点开始，也总可以无歧义地指出字符编码的起始和结束位置。UTF-8 首字节和后续字节总有区别，因此一个编码不能从另一个编码的中间开始。首字节确定编码总长度，因此没有编码可以是其他编码的前缀。这样也相应带来了很多好处。比如，从 UTF-8 字符串中搜索一个 ASCII 定界符只需扫描定界符的字节即可。这个定界符不可能是其他多字节编码中的一个字节，因此根本无须考虑 UTF-8 的结构。类似地，从一个字符串中搜索另一个字节字符串的算法无须修改 UTF-8 字符串，有的甚至都不需要检查被搜索文本的所有字节。

虽然可变宽度编码比固定宽度编码更复杂，但以上特点让 UTF-8 比想象得更容易使用。Rust 标准库为我们处理了大多数的复杂细节。

17.1.3 文本方向性

有些文字是从左向右书写的，比如拉丁文、西里尔文、泰文。而有些文字是从右向左书写的，比如希伯来文、阿拉伯文。Unicode 按照文字正常情况下书写或阅读的顺序存储字符。因此对于希伯来文而言，字符串的首字节保存的是要写在最右边的字符的编码：

```
assert_eq!("ערב טוב".chars().next(), Some('ע'));
```

标准库中有些方法使用 `left` 和 `right` 表示文本的开始和结束。在描述这些函数时，我们会详细说明它们真正的意图。

17.2 字符（char）

Rust 的 `char` 类型是保存 Unicode 码点的 32 位值。`char` 一定会落在 `0` 到 `0xd7ff` 或者 `0xe000` 到 `0x10ffff` 的范围内。所有用于创建和操作 `char` 值的方法都会确保这一点。`char` 类型实现了 `Copy` 和 `Clone`，以及比较、散列、格式的所有常用特型。

在下文的介绍中，变量 `ch` 的类型就是 `char`。

17.2.1 字符分类

`char` 类型有一些方法将字符分为几个常见的类别。这些类别都源自 Unicode，如表 17-1 所示。

表17-1：检测字符类别的方法

方　　法	简　　介	示　　例

`ch.is_numeric()`	数值字符，包括 Unicode 普通类别“Number; digit”和“Number; letter”，但不包括“Number; other”	`'4'.is_numeric()`
		`'ᛮ'.is_numeric()`
		`!'⑧'.is_numeric()`

`ch.is_alphabetic()`	字母字符，包括 Unicode 的“Alphabetic”派生属性	`'q'.is_alphabetic()` `'七'.is_alphabetic()`

`ch.is_alphanumeric()`	数值或字母字符，包括上面两个类别	`'9'.is_alphanumeric()`
		`'饂'.is_alphanumeric()`
		`!'*'.is_alphanumeric()`

`ch.is_whitespace()`	空白字符，包括 Unicode 字符属性“WSpace=Y”	`' '.is_whitespace()`
		`'\n'.is_whitespace()`
		`'\u{A0}'.is_whitespace()`

`ch.is_control()`	控制字符，包括 Unicode 的“Other, control”普通类别	`'\n'.is_control()` `'\u{85}'.is_control()`

17.2.2 处理数字

处理数字有以下几种方法。

- **`ch.to_digit(radix)`** 决定 `ch` 是否是基数为 `radix` 的数字。如果是，就返回 `Some(num)`，其中 `num` 是 `u32`。否则，返回 `None`。这个方法仅适用于 ASCII 数字，而不适用于 `char::is_numeric` 所适用的更广泛的字符类。`radix` 参数的范围是从 2 到 36。如果

radix 大于 10，那么 ASCII 字母（无论大小写）将作为值为 10 到 35 的数字。

- 自由函数 **std::char::from_digit(num, radix)** 可以把 u32 数字值 num 转换为 char（如果可以的话）。如果 num 可以用 radix 基数下的一个数字表示，那么 from_digit 返回 Some(ch)，其中 ch 就是数字。如果 radix 大于 10，ch 就可以是小写字母。否则，返回 None。

  执行反向操作的方法是 to_digit。如果 std::char::from_digit(num, radix) 是 Some(ch)，那么 ch.to_digit(radix) 就是 Some(num)。如果 ch 是 ASCII 数字或小写字母，则反之亦然。

- **ch.is_digit(radix)** 在 ch 是基数为 radix 下的 ASCII 数字时返回 true。这个方法等价于 ch.to_digit(radix) != None。

好，来看几个例子：

```
assert_eq!('F'.to_digit(16), Some(15));
assert_eq!(std::char::from_digit(15, 16), Some('f'));
assert!(char::is_digit('f', 16));
```

17.2.3　字符大小写转换

有关字符大小写的方法如下。

- **ch.is_lowercase()** 和 **ch.is_uppercase()** 表示 ch 是小写字母字符还是大写字母字符。它们遵循 Unicode 的 Lowercase 和 Uppercase 派生属性，因此这两个方法也适用于非拉丁字母，如希腊字母和西里尔字母，当然对 ASCII 字母同样有用。
- **ch.to_lowercase()** 和 **ch.to_uppercase()** 返回迭代器，根据 Unicode 的 Default Case Conversion 算法生成 ch 对应的小写或大写字符：

  ```
  let mut upper = 's'.to_uppercase();
  assert_eq!(upper.next(), Some('S'));
  assert_eq!(upper.next(), None);
  ```

  这两个方法之所以返回迭代器而不是字符，是因为 Unicode 中的大小写转换不一定是一对一的过程：

  ```
  // 德语字母"sharp S"的大写形式是"SS"：
  let mut upper = 'ß'.to_uppercase();
  assert_eq!(upper.next(), Some('S'));
  assert_eq!(upper.next(), Some('S'));
  assert_eq!(upper.next(), None);

  // Unicode中Turkish的带点大写'İ'转换为小写形式是'i'后跟'\u{307}'，
  // 带上点，这样再转换回大写才会保留上面的点
  let ch = 'İ'; // `'\u{130}'`
  let mut lower = ch.to_lowercase();
  assert_eq!(lower.next(), Some('i'));
  assert_eq!(lower.next(), Some('\u{307}'));
  assert_eq!(lower.next(), None);
  ```

为方便起见，这些迭代器都实现了 std::fmt::Display 特型，因此可以直接将它们传给 println! 或 write! 宏。

17.2.4 与整数相互转换

Rust 的 as 操作符可以把 char 转换为任何整数类型，高位会被静默屏蔽：

```
assert_eq!('B' as u32, 66);
assert_eq!('饂' as u8, 66);   // 高位被截掉了
assert_eq!('二' as i8, -116); // 同上
```

as 操作符可以将任何 u8 值转换为 char，而 char 也实现了 From<u8>。不过，更宽的整数类型可以表示无效码点，因此必要时应该使用 std::char::from_u32，这个方法返回 Option<char>：

```
assert_eq!(char::from(66), 'B');
assert_eq!(std::char::from_u32(0x9942), Some('饂'));
assert_eq!(std::char::from_u32(0xd800), None); // 为UTF-16保留
```

17.3 String与str

Rust 的 String 和 str 类型确保只保存格式良好的 UTF-8。为此，标准库通过限制创建 String 和 str 值的方式以及对它们可以执行的操作来确保这一点。这样，这两个值在引入时是格式良好的，使用时也是如此。相关的方法也都会保证安全的操作不会引入格式错误的 UTF-8。这也简化了操作文本的代码。

Rust 将文本处理方法放在 str 还是 String 上，取决于该方法是需要可伸缩缓冲区还是只需要就地操作文本。因为 String 解引用为 &str，所以 str 上定义的所有方法也都可以在 String 上直接调用。本节会介绍这两个类型上的所有方法，按大致的功能分组。

文本处理相关的方法按照字节偏移量来索引文本，也按字节来度量长度，并不按字符。实践中，考虑到 Unicode 的特点，按字符索引并不像看起来那么有用。反而按字节偏移量索引更快也更简单。如果要使用的字节偏移量恰好落在某个字符 UTF-8 编码的中间，方法则会诧异，因此不能以这种方式引入格式错误的 UTF-8。

String 实际上是 Vec<u8> 的包装类型，同时确保向量内容始终是格式良好的 UTF-8。Rust 不会修改 String 去使用更复杂的表示，因此可以认为 String 与 Vec 性能相同。

在接下来的介绍中，相关变量与对应类型如表 17-2 所示。

表17-2：变量与类型

变　　量	推测类型
string	String
slice	&str 或解引用为 &str 的类型，比如 String 或 Rc<String>
ch	char
n	usize，长度
i、j	usize，字节偏移量
range	usize 字节偏移量范围，可能是全限定（i..j）、部分限定（i..、..j）或无限定（..）
pattern	任意模式类型：char、String、&str、&[char] 或 FnMut(char) -> bool

17.3.6 节将讨论模式类型。

17.3.1　创建字符串值

以下是几种常见的创建 String 值的方式。

- **String::new()** 返回全新的空字符串，此时没有分配在堆上的缓冲区，后续会根据需要分配。
- **String::with_capacity(n)** 返回全新的空字符串，同时堆上会预分配至少能容纳 n 字节的缓冲区。如果事先知道要构建的字符串的长度，使用这个构造函数可以一开始就得到大小正确的缓冲区，从而避免后续构建字符串时再调整缓冲区大小。当然，如果字符串长度超过 n 字节，那其缓冲区仍会增大。与向量类似，字符串也有 capacity、reserve 和 shrink_to_fit 方法，不过一般来说默认的分配逻辑就可以了。
- **slice.to_string()** 分配一个全新的 String，其内容是 slice 的副本。本书已经多次用到类似 "literal text".to_string() 这样的表达式通过字符串字面量创建 String 了。
- **iter.collect()** 通过拼接迭代器的所有项(可以是 char、&str 或 String 值)来构建字符串。比如，要删除一个字符串中的所有空格，可以这样写：

```
let spacey = "man hat tan";
let spaceless: String =
    spacey.chars().filter(|c| !c.is_whitespace()).collect();
assert_eq!(spaceless, "manhattan");
```

  这样使用 collect 是利用了 String 对 std::iter::FromIterator 特型的实现。

- &str 类型不能实现 Clone，这个特型要求对 &T 的克隆返回一个 T 类型的值，但 str 是非固定大小的。不过，&str 实现了 ToOwned，实现者可以指定自己拥有的等价值，因此 **slice.to_owned()** 将 slice 的副本作为一个全新分配的 String 返回。

17.3.2　简单检查

以下方法可以从字符串切片获得基本信息。

- **slice.len()** 返回以字节计的 slice 的长度。
- **slice.is_empty()** 在 slice.len() == 0 时返回 true。
- **slice[range]** 返回借用 slice 中指定部分的切片。部分限定和无限定范围都可以，比如：

```
let full = "bookkeeping";
assert_eq!(&full[..4], "book");
assert_eq!(&full[5..], "eeping");
assert_eq!(&full[2..4], "ok");
assert_eq!(full[..].len(), 11);
assert_eq!(full[5..].contains("boo"), false);
```

- 不能像 slice[i] 一样仅用一个位置索引字符串切片。从给定的字节偏移值取得一个字符有点麻烦。必须先基于切片产生一个 chars 迭代器，让迭代器解析出相应字符的 UTF-8：

```
let parenthesized = "Rust (饂)";
assert_eq!(parenthesized[6..].chars().next(), Some('饂'));
```

  好在很少需要这样做。Rust 也提供了迭代切片的更便捷方式，17.3.8 节会介绍。

- **slice.split_at(i)** 返回从 slice 借用的两个共享切片的元组，第一个截止到字节偏移值 i，第二个从 i 之后开始。换句话说，就是返回 (slice[..i], slice[i..])。
- **slice.is_char_boundary(i)** 在字节偏移值 i 落在字符边界上时返回 true，因此适合作为 slice 的偏移值。

自然地，切片可以比较相等、顺序和散列。比较顺序只是简单地将字符串看成 Unicode 码点的序列，并按照字典顺序对它们进行比较。

17.3.3 追加和插入文本

以下方法可以向 String 中添加文本。

- **string.push(ch)** 会把字符 ch 追加到 string 末尾。
- **string.push_str(slice)** 会追加 slice 的全部内容。
- **string.extend(iter)** 将迭代器 iter 生成的所有项追加到字符串。迭代器可以生成 char、str 或 String 值。这些都是 String 对 std::iter::Extend 的实现。

```
let mut also_spaceless = "con".to_string();
also_spaceless.extend("tri but ion".split_whitespace());
assert_eq!(also_spaceless, "contribution");
```

- **string.insert(i, ch)** 会在字节偏移值 i 的位置向 string 中插入字符 ch。这会导致 i 之后的所有字符移位以便给 ch 腾出位置。因此以这种方式构建字符串需要的时间与字符串长度的平方成正比。
- **string.insert_str(i, slice)** 与 insert 一样，只不过是插入 slice 的内容，当然也同样有性能问题。

String 实现了 std::fmt::Write，也就意味着 write! 和 writeln! 宏可以给 String 追加格式化文本：

```
use std::fmt::Write;

let mut letter = String::new();
writeln!(letter, "Whose {} these are I think I know", "rutabagas")?;
writeln!(letter, "His house is in the village though;")?;
assert_eq!(letter, "Whose rutabagas these are I think I know\n\
                    His house is in the village though;\n");
```

因为 write! 和 writeln! 是用于写入输出流的，所以它们返回一个 Result，如果不处理错误 Rust 会编译不通过。上面的代码使用 ? 操作符来处理了错误，但写入 String 是不会出错的，因此在这里调用 .unwrap() 也可以。

由于 String 实现了 Add<&str> 和 AddAssign<&str>，因此可以这样编写代码：

```
let left = "partners".to_string();
let mut right = "crime".to_string();
assert_eq!(left + " in " + &right, "partners in crime");

right += " doesn't pay";
assert_eq!(right, "crime doesn't pay");
```

在操作数为字符串时，+ 操作符会取得其左操作数的值，因此实际上可以重用相加得到的结果 String。而且，如果左操作数的缓冲区足够大以保存结果，就不会发生重新分配。

很可惜这里少了对称性，+ 的左操作数不能是 &str，因此不能这样用：

```
let parenthetical = "(" + string + ")";
```

而必须这样：

```
let parenthetical = "(".to_string() + &string + ")";
```

不过，这一限制确实导致很难从末尾反方向构建字符串。这种方法性能很差，因为文本必须朝缓冲区末尾重复移动。

可是以向尾部追加小片段的方式构建字符串则性能很好。String 的行为类似向量，在需要更大容量时至少会将缓冲区扩大一倍。正如 3.4.3 节所解释的，重新复制的开销取决于最终大小。即便如此，使用 String::with_capacity 从一开始就创建缓冲区大小合适的字符串，可以避免重新分配，减少调用堆分配程序的次数。

17.3.4 删除文本

String 有几个删除文本的方法（不影响字符串的容量。如果需要释放内存，可以使用 shrink_to_fit）。

- **string.clear()** 将 string 重置为空字符串。
- **string.truncate(n)** 会丢弃字节偏移值 n 之后的所有字符，导致 string 的长度最大为 n。如果 string 不足 n 字节，调用此方法则没有效果。
- **string.pop()** 从 string 中删除最后一个字符（前提是有字符），并以 Option<char> 方式返回。
- **string.remove(i)** 从 string 中删除字节偏移值 i 所在的字符并返回该字符，后面的字符向前移动。所花时间与后面字符的数量成正比。
- **string.drain(range)** 根据给定字节索引的范围返回迭代器，并且在迭代器被清除时删除相应字符。删除范围后，范围之后的字符向前移动：

```
let mut choco = "chocolate".to_string();
assert_eq!(choco.drain(3..6).collect::<String>(), "col");
assert_eq!(choco, "choate");
```

  如果只想删除范围，那么马上清除迭代器，不从中取值即可：

```
let mut winston = "Churchill".to_string();
winston.drain(2..6);
assert_eq!(winston, "Chill");
```

17.3.5 搜索与迭代的约定

Rust 标准库中与搜索和迭代文本相关的函数，在命名上遵循了一些约定，以方便记忆。

- 大多数操作从头到尾处理文本，但名字以 r 开头的操作从后向前处理。比如，rsplit 是 split 的从后向前的版本。某些情况下，改变处理方向不仅会影响产生值的顺序，也会影响值本身。具体的例子参见图 17-3。

- 迭代器的名字如果以 `n` 结尾，就表示会对自己限定匹配的次数。
- 迭代器的名字如果以 **`_indices`** 结尾，表示会产生它们在切片中的字节偏移量，以及通常可迭代的值。

标准库没有提供所有操作的全部组合。比如，很多操作不需要带 `n` 的变体，提前结束迭代也非常简单。

17.3.6 搜索文本的模式

当标准库函数需要搜索（search）、匹配（match）、分割（split）或修剪（trim）文本时，它接收几种不同类型的参数来表示要查找的内容：

```
let haystack = "One fine day, in the middle of the night";

assert_eq!(haystack.find(','), Some(12));
assert_eq!(haystack.find("night"), Some(35));
assert_eq!(haystack.find(char::is_whitespace), Some(3));
```

这些类型被称作**模式**（pattern），且大多数操作支持它们：

```
assert_eq!("## Elephants"
           .trim_left_matches(|ch: char| ch == '#' || ch.is_whitespace()),
           "Elephants");
```

标准库支持 4 种主要的模式。

- `char` 作为模式用于匹配字符。
- `String`、`&str` 或 `&&str` 作为模式用于匹配等于模式的子字符串。
- `FnMut(char) -> bool` 闭包作为模式用于匹配闭包返回 `true` 的一个字符。
- `&[char]` 作为模式（不是 `&str`,而是 `char` 值的切片）用于匹配出现在列表中的任一字符。注意，如果你将列表作为数组字面量写出来，那可能需要使用 `as` 表达式将其转换为正确的类型。

  ```
  let code = "\t    function noodle() { ";
  assert_eq!(code.trim_left_matches(&[' ', '\t'] as &[char]),
             "function noodle() { ");
  // 更短的写法：&\[' ', '\t'][..]
  ```

  否则，Rust 在看到固定大小的数组类型 `&[char; 2]` 时会不知所措，因为这不是模式类型。

在标准库自己的代码中，模式可以是实现 `std::str::Pattern` 特型的任何类型。`Pattern` 的具体细节还没有稳定，因此在稳定版的 Rust 中还不允许自定义类型实现它。不过，这扇门已经对将来的正则表达式和其他复杂模式打开了。Rust 保证当前支持的模式类型在将来仍然有效。

17.3.7 搜索与替换

Rust 提供了几个按模式搜索切片中的目标的方法，有些方法还支持用新文本替换目标。

- **`slice.contains(pattern)`** 在 `slice` 包含与 `pattern` 匹配的内容时返回 `true`。

- **`slice.starts_with(pattern)`** 和 **`slice.ends_with(pattern)`** 在 slice 的初始或最终文本与 pattern 匹配时返回 true。

```
assert!("2017".starts_with(char::is_numeric));
```

- **`slice.find(pattern)`** 和 **`slice.rfind(pattern)`** 在 slice 包含匹配 pattern 的内容时返回 Some(i)，其中 i 是匹配项的字节偏移量。find 方法返回第一个匹配，rfind 方法返回最后一个匹配。

```
let quip = "We also know there are known unknowns";
assert_eq!(quip.find("know"), Some(8));
assert_eq!(quip.rfind("know"), Some(31));
assert_eq!(quip.find("ya know"), None);
assert_eq!(quip.rfind(char::is_uppercase), Some(0));
```

- **`slice.replace(pattern, replacement)`** 返回以 replacement 替换所有匹配 pattern 的内容之后得到的新 String。

```
assert_eq!("The only thing we have to fear is fear itself"
           .replace("fear", "spin"),
           "The only thing we have to spin is spin itself");

assert_eq!("`Borrow` and `BorrowMut`"
           .replace(|ch:char| !ch.is_alphanumeric(), ""),
           "BorrowandBorrowMut");
```

- **`slice.replacen(pattern, replacement, n)`** 同上，只是最多替换前 n 个匹配项。

17.3.8 迭代文本

Rust 标准库提供了几种迭代切片文本的方式。图 17-3 展示了其中几个例子。

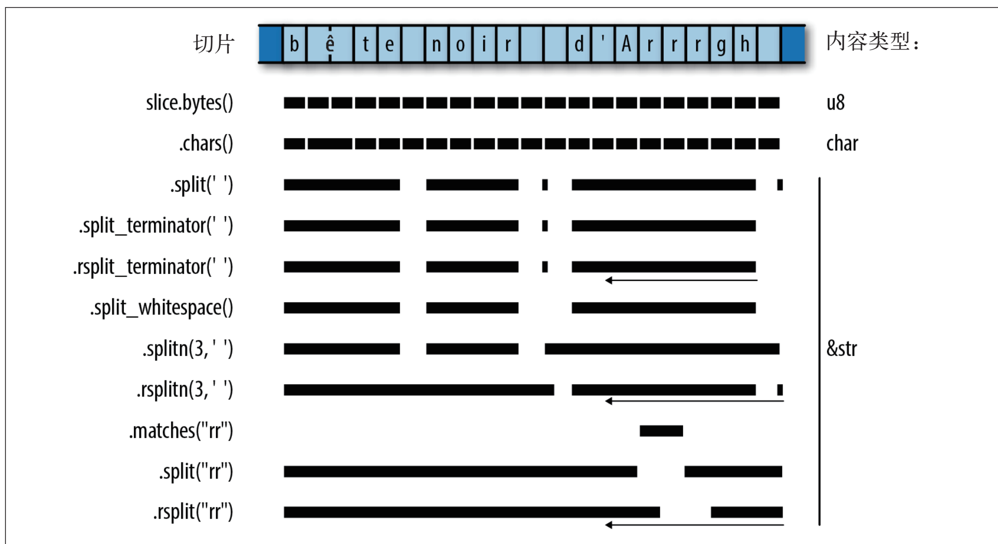


图 17-3：迭代切片的一些方式

split 和 match 系列方法是相互补充的，因为分割的结果其实就是匹配项之间的范围。

对某些模式而言，从后向前搜索可能会改变产生的值。比如，图 17-3 中基于模式 "rr" 的分割。始终匹配一个字符的模式不会出现这个结果。如果一个迭代器能够在任何方向都产生相同的匹配项集合（即只有顺序不同），则这个迭代器就是一个 DoubleEndedIterator，意味着可以使用其 rev 方法按另一个顺序迭代，即可以从任何一端取值。

- **slice.chars()** 基于 slice 的字符返回一个迭代器。
- **slice.char_indices()** 基于 slice 的字符及它们的字节偏移量返回一个迭代器。

```
assert_eq!("élan".char_indices().collect::<Vec<_>>(),
           vec![(0, 'é'), // 有一个双字节UTF-8编码
                (2, 'l'),
                (3, 'a'),
                (4, 'n')]);
```

  注意，这跟 .chars().enumerate() 不同，因为它会提供每个字符在切片中的字节偏移量，而不仅仅是字符编号。

- **slice.bytes()** 基于 slice 中的个别字节返回一个迭代器，暴露 UTF-8 编码。

```
assert_eq!("élan".bytes().collect::<Vec<_>>(),
           vec![195, 169, b'l', b'a', b'n']);
```

- **slice.lines()** 基于 slice 中的行返回一个迭代器。行的终止符是 "\n" 或 "\r\n"。这个迭代器产生的值是从 slice 借用的 &str。另外，产生的值不包含行终止符。
- **slice.split(pattern)** 基于按照 pattern 分割 slice 得到的部分返回一个迭代器。两个相邻的匹配或者与 slice 开头、结尾的匹配都会返回空字符串。
- **slice.rsplit(pattern)** 同上，只不过是从后向前扫描 slice，而且按该顺序进行匹配。
- **slice.split_terminator(pattern)** 和 **slice.rsplit_terminator(pattern)** 与上一个方法类似，只不过 pattern 被当成终止符而不是分隔符。如果 pattern 恰好匹配 slice 的两头，那么迭代器不会（像 split 和 rsplit 那样）生成表示匹配与切片两头之间空字符串的空切片。例如：

```
// ':'字符在这里是分隔符，注意最后的""
assert_eq!("jimb:1000:Jim Blandy:".split(':').collect::<Vec<_>>(),
           vec!["jimb", "1000", "Jim Blandy", ""]);

// '\n'字符在这里是终止符
assert_eq!("127.0.0.1 localhost\n\
            127.0.0.1 www.reddit.com\n"
           .split_terminator('\n').collect::<Vec<_>>(),
           vec!["127.0.0.1 localhost",
                "127.0.0.1 www.reddit.com"]);
                // 注意，最后没有""!
```

- **slice.splitn(n, pattern)** 和 **slice.rsplitn(n, pattern)** 跟 split 和 rsplit 类似，只不过最多把字符串分割成 n 个切片，从 pattern 的第 1 次匹配到第 n-1 次匹配。

- **slice.split_whitespace()** 基于空白将 slice 分隔的部分返回一个迭代器。连续多个空白字符作为一个分隔符。末尾的空白会被忽略。这里**空白**的定义与 char::is_whitespace 中相同。

```
let poem = "This is just to say\n\
            I have eaten\n\
            the plums\n\
            again\n";

assert_eq!(poem.split_whitespace().collect::<Vec<_>>(),
           vec!["This", "is", "just", "to", "say",
                "I", "have", "eaten", "the", "plums",
                "again"]);
```

- **slice.matches(pattern)** 基于 pattern 在 slice 中找到的匹配项返回一个迭代器。**slice.rmatches(pattern)** 也一样，只不过是从后向前迭代。
- **slice.match_indices(pattern)** 和 **slice.rmatch_indices(pattern)** 跟前面的方法类似，只不过产生的值是 (offset, match) 对，其中 offset 是匹配开始位置的字节偏移量，match 是匹配的切片。

17.3.9　修剪

修剪（trim）字符串就是从字符串的开头和末尾去掉内容（通常是空白符）。修剪常用于清理从文件中读到的带缩进的文本，或者一行末尾意外带着的空白符，以便让结果更清晰。

- **slice.trim()** 返回 slice 的子切片，不包含开头和末尾的空白符。**slice.trim_left()** 只省略开头的空白符，**slice.trim_right()** 只省略末尾的空白符。

```
assert_eq!("\t*.rs  ".trim(), "*.rs");
assert_eq!("\t*.rs  ".trim_left(), "*.rs  ");
assert_eq!("\t*.rs  ".trim_right(), "\t*.rs");
```

- **slice.trim_matches(pattern)** 返回 slice 的子切片，不包含开头和末尾匹配 pattern 的内容。**trim_left_matches** 和 **trim_right_matches** 方法仅对开头或末尾的匹配执行同样的操作。

```
assert_eq!("001990".trim_left_matches('0'), "1990");
```

注意，以上方法名中的 left 和 right 分别指的是切片的开头和末尾，与切片中文本的方向无关。

17.3.10　字符串大小写转换

方法 **slice.to_uppercase()** 和 **slice_to_lowercase()** 返回新分配的字符串，其保存着转换为大写或小写之后的 slice 文本。结果的长度不一定与 slice 相同，参见 17.2.3 节。

17.3.11　从字符串解析出其他类型

Rust 为从字符串中解析值和产生值的文本化表示提供了标准的特型。

如果一个类型实现了 std::str::FromStr 特型，那么它就拥有从字符串切片中解析值的标准方法：

```
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

所有常见的机器类型都实现了 FromStr：

```
use std::str::FromStr;

assert_eq!(usize::from_str("3628800"), Ok(3628800));
assert_eq!(f64::from_str("128.5625"), Ok(128.5625));
assert_eq!(bool::from_str("true"), Ok(true));

assert!(f64::from_str("not a float at all").is_err());
assert!(bool::from_str("TRUE").is_err());
```

用于存储 IPv4 或 IPv6 互联网地址的枚举（enum）类型 std::net::IpAddr 也实现了 FromStr：

```
use std::net::IpAddr;

let address = IpAddr::from_str("fe80::0000:3ea9:f4ff:fe34:7a50")?;
assert_eq!(address,
           IpAddr::from([0xfe80, 0, 0, 0, 0x3ea9, 0xf4ff, 0xfe34, 0x7a50]));
```

字符串切片有一个 parse 方法，其可以将切片解析为你想要的任何类型，可以假设它实现了 FromStr。与 Iterator::collect 一样，有时候可能需要写出想要的类型，因此 parse 并不总是比直接调用 from_str 更清晰：

```
let address = "fe80::0000:3ea9:f4ff:fe34:7a50".parse::<IpAddr>()?;
```

17.3.12　将其他类型转换为字符串

将非文本值转换为字符串主要有 3 种方式。

- 具有人类可读的自然打印形式的类型可以实现 std::fmt::Display 特型，这样就可以在 format! 宏中使用 {} 格式说明符了：

```
assert_eq!(format!("{}, wow", "doge"), "doge, wow");
assert_eq!(format!("{}", true), "true");
assert_eq!(format!("({:.3}, {:.3})", 0.5, f64::sqrt(3.0)/2.0),
           "(0.500, 0.866)");

// 使用上面的address
let formatted_addr: String = format!("{}", address);
assert_eq!(formatted_addr, "fe80::3ea9:f4ff:fe34:7a50");
```

  与字符、字符串和切片一样，Rust 的所有机器数值类型都实现了 Display。对于智能指针类型而言，如果 T 实现 Display，则 Box<T>、Rc<T> 和 Arc<T> 也就实现了：它们打印出来的形式就是它们引用目标的形式。Vec 和 HashMap 等容器没有实现 Display，因为这些类型没有唯一人类可读的自然打印形式。

- 如果一个类型实现 Display，则标准库自动为其实现 std::str::ToString 特型，这个特型唯一的方法 to_string 有时候比更灵活的 format! 更方便：

  ```
  // 继续上面的例子
  assert_eq!(address.to_string(), "fe80::3ea9:f4ff:fe34:7a50");
  ```

  ToString 特型出现的时间早于 Display，而且没那么灵活。对于自定义类型来说，通常应该实现 Display 而非 ToString。

- 标准库中的每个公共类型都实现了 std::fmt::Debug，这个特型可以接收一个值并将其格式化为在某种程度上对程序员有帮助的字符串形式。使用 Debug 生成字符串最简单的方法就是借助 format! 宏的 {:?} 格式说明符：

  ```
  // 继续上面的例子
  let addresses = vec![address,
                       IpAddr::from_str("192.168.0.1")?];
  assert_eq!(format!("{:?}", addresses),
             "[V6(fe80::3ea9:f4ff:fe34:7a50), V4(192.168.0.1)]");
  ```

  这里利用了 Vec<T> 在 T 实现 Debug 的情况下对 Debug 的包装实现。Rust 的所有集合类型都有这种实现。

  自定义类型也应该实现 Debug。通常，最好是让 Rust 派生一个实现，就像下面这个 Complex 类型一样：

  ```
  #[derive(Copy, Clone, Debug)]
  struct Complex { r: f64, i: f64 }
  ```

Display 和 Debug 格式化特型只是 format! 宏及其将值格式化为文本的众多示例中的两个。17.4 节将介绍其他特型以及如何实现它们。

17.3.13 作为其他类文本类型借用

可以通过以下两种方法借用切片的内容。

- 切片和 String 实现了 AsRef<str>、AsRef<[u8]>、AsRef<Path> 和 AsRef<OsStr>。很多标准库函数使用这些特型作为自己参数类型的绑定，因此可以直接将切片或字符串传给它们，即使这些函数需要的是其他类型。关于这些特型的详细解释，参见 13.7 节。
- 切片和 String 也实现了 std::borrow::Borrow<str> 特型。HashMap 和 BTreeMap 使用 Borrow 让 String 可以作为表中的键。详细内容参见 13.8 节。

17.3.14 访问UTF-8格式的文本

获取字节表示的文本有两种主要方式，这取决于你是想取得字节的所有权，还是仅仅想借用它们。

- **slice.as_bytes()** 借用 slice 的字节作为 &[u8]。因为这不是一个可修改引用，所以 slice 可以假设其字节会保持为格式良好的 UTF-8。

- **string.into_bytes()** 取得 string 的所有权并按值返回这个字符串字节的 Vec<u8>。这个转换代价很小，因为它只是把字符串用作缓冲区的 Vec<u8> 转了一道手。由于 string 不再存在，因此获取的字节没必要再是格式良好的 UTF-8，调用者可以按照自己的需要修改这个 Vec<u8>。

17.3.15　从UTF-8数据产生文本

如果有一段包含 UTF-8 数据的字节值，那么根据如何处理错误，有以下几种方式可以将其转换为 String 或切片。

- **str::from_utf8(byte_slice)** 接收一个 &[u8] 字节切片，返回一个 Result：如果 byte_slice 包含格式良好的 UTF-8，则返回 Ok(&str)，否则返回错误。
- **String::from_utf8(vec)** 尝试基于传入的 Vec<u8> 值构建一个字符串。如果 vec 保存着格式良好的 UTF-8，from_utf8 就返回 Ok(string)，其中 string 是取得 vec 所有权并将其作为缓冲的字符串。这个过程不涉及分配堆内存或文本复制。

如果字节不是格式良好的 UTF-8，则这个方法返回 Err(e)，其中 e 是一个 FromUtf8Error 错误值。此时调用 e.into_bytes() 会得到原始的向量 vec，即转换失败不会丢失值：

```
let good_utf8: Vec<u8> = vec![0xe9, 0x8c, 0x86];
assert_eq!(String::from_utf8(good_utf8).ok(), Some("錆".to_string()));

let bad_utf8: Vec<u8> = vec![0x9f, 0xf0, 0xa6, 0x80];
let result = String::from_utf8(bad_utf8);
assert!(result.is_err());
// 因为String::from_utf8失败，所以它并未消费原始的向量，
// 通过错误值可以再拿回未受影响的向量
assert_eq!(result.unwrap_err().into_bytes(),
           vec![0x9f, 0xf0, 0xa6, 0x80]);
```

- **String::from_utf8_lossy(byte_slice)** 尝试基于字节的共享切片 &[u8] 构建一个 String 或 &str。这个转换一定会成功，格式不正确的 UTF-8 会被 Unicode 替换字符取代。返回的值是一个 Cow<str>。如果 byte_slice 包含格式良好的 UTF-8，Cow<str> 就直接从 byte_slice 借用一个 &str；如果 byte_slice 包含格式不正确的 UTF-8，Cow<str> 就拥有一个新分配的 String，格式不正确的字节会被替换字符取代。因此，byte_slice 如果包含格式良好的 UTF-8，就不会发生堆内存分配或复制操作。下一节将详细谈论 Cow<str>。
- 如果你知道 Vec<u8> 包含格式良好的 UTF-8，就可以调用不安全的函数 **String::from_utf8_unchecked**。这个方法会简单地将 Vec<u8> 包装为一个 String 并返回它，根本不检查字节的格式。换句话说，调用这个方法的前提是你有责任确保没有将格式错误的 UTF-8 引入系统，这就是这个函数被标记为 unsafe 的原因。
- 类似地，**str::from_utf8_unchecked** 接收一个 &[u8] 并将其返回为一个 &str，同样不检查字节的格式是不是格式良好的 UTF-8。与 String::from_utf8_unchecked 一样，你有责任确保这个转换是安全的。

17.3.16 阻止分配

假设你想写一个问候用户的程序。在 Unix 上，可以这样写：

```
fn get_name() -> String {
    std::env::var("USER") // Windows使用"USERNAME"
        .unwrap_or("whoever you are".to_string())
}

println!("Greetings, {}!", get_name());
```

对 Unix 用户而言，这个程序会按照用户名问候他们。而对 Windows 用户而言，这样拿不到用户名，因此其提供了一个后备文本。

std::env::var 函数返回一个 String，当然很可能不会返回用户名。这就意味着后备文本必须也作为 String 返回。让人不满意的是，get_name 在返回静态字符串时，根本不应该有内存分配。

问题的核心在于有时候 get_name() 返回的值可能是所有型的 String，有时候也可能是 &'static str'。到底是哪一个，只有到运行程序的时候才知道。这种动态的特点提示我们可以考虑使用 std::borrow::Cow，这个"写时克隆"类型既可以保存所有型数据，也可以保存借用的数据。

正如 13.11 节所解释的，Cow<'a, T> 是一个枚举，其包含两个变体：Borrowed 和 Owned。Borrowed 保存一个引用 &'a T，Owned 保存 &T 的所有型版本：对 &str 是 String，对 &[i32] 是 Vec<i32>，等等。无论是 Borrowed 还是 Owned，Cow<'a, T> 始终可以生成一个 &T 供你使用。事实上，Cow<'a, T> 解引用为 &T，有点像智能指针。

像下面这样把 get_name 修改为返回 Cow：

```
use std::borrow::Cow;

fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| Cow::Owned(v))
        .unwrap_or(Cow::Borrowed("whoever you are"))
}
```

如果读取 "USER" 环境变量成功，则 map 将得到的字符串作为 Cow::Owned 返回。如果失败，unwrap_or 将其静态的 &str 作为 Cow::Borrowed 返回。调用代码不变：

```
println!("Greetings, {}!", get_name());
```

只要 T 实现 std::fmt::Display 特型，显示 Cow<'a, T> 就会得到与显示 T 一样的结果。

Cow 在可能需要也可能不需要修改你借用的某个文本时也很有用。在不需要修改的时候，可以继续借用它。而 Cow 这个名字代表的 Clone-on-write（写时克隆）行为意味着也可以根据需要返回一个值的所有型、可修改的副本。Cow 的 to_mut 方法确保 Cow 是 Cow::Owned，必要时会应用值的 ToOwned 实现，然后返回这个值的可修改引用。

因此，如果你发现有些用户（不是全部）也有称谓想被提及，那可以这样：

```
fn get_title() -> Option<&'static str> { ... }

let mut name = get_name();
if let Some(title) = get_title() {
    name.to_mut().push_str(", ");
    name.to_mut().push_str(title);
}

println!("Greetings, {}!", name);
```

这个可能会产生如下输出：

```
$ cargo run
Greetings, jimb, Esq.!
$
```

关键在于，此时如果 get_name() 返回一个静态字符串，而 get_title() 返回 None，Cow 就会携带一个静态字符串，一直到调用 println!。这样，就做到了代码很直观，同时也只在必要时才会分配内存。

因为 Cow 频繁用于字符串，所以标准库为 Cow<'a, str> 提供了一些特殊支持。比如，它提供了来自 String 和 &str 的 From 和 Into 转换，因此 get_name 还可以写得更简洁：

```
fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| v.into())
        .unwrap_or("whoever you are".into())
}
```

Cow<'a, str> 也实现了 std::ops::Add 和 std::ops::AddAssign，因此给名字添加称谓还可以这样写：

```
if let Some(title) = get_title() {
    name += ", ";
    name += title;
}
```

或者，由于 String 可以作为 write! 宏的目标，因此这样写也行：

```
use std::fmt::Write;

if let Some(title) = get_title() {
    write!(name.to_mut(), ", {}", title).unwrap();
}
```

跟以前一样，除非要修改 Cow，否则不会分配内存。

要注意，不是所有 Cow<..., str> 都必须是 'static。在需要复制之前，可以一直使用 Cow 借用之前计算的文本。

17.3.17　字符串作为泛型集合

String 实现了 std::default::Default 和 std::iter::Extend：default 返回一个空字符串，而 extend 可以向一个字符串末尾追加字符、字符串切片或字符串。这跟 Rust 中其他集合类型

（比如 Vec 和 HashMap）为 collect 和 partition 泛型构建模式实现的特型组合是一样的。

&str 类型也实现了 Default，返回一个空切片。这在某些边界情况下很有用。比如，这样就可以为包含字符串切片的结构派生 Default。

17.4　格式化值

本书中很多示例用到了文本格式化宏，比如 println!：

```
println!("{:.3}μs: relocated {} at {:#x} to {:#x}, {} bytes",
         0.84391, "object",
         140737488346304_usize, 6299664_usize, 64);
```

以上调用会生成如下输出：

```
0.844μs: relocated object at 0x7fffffffdcc0 to 0x602010, 64 bytes
```

这里的字符串字面量充当输出的模板。模板中的每个 {...} 都会被后面某个参数的格式化形式所取代。模板字符串必须是常量，这样 Rust 就可以在编译时根据参数类型检查它。每个参数必须都用上，否则 Rust 会报编译时错误。

有几个标准库特性中都用到了这种模板语言来格式化字符串：

- format! 宏使用模板来构建 String；
- println! 和 print! 宏将格式化后的文本写入标准输出流；
- writeln! 和 write! 宏将格式化后的文本写入指定输出流；
- panic! 宏使用模板构建一个（可能包含有用信息的）终止诧异的表达式。

Rust 的格式化功能是开放式的。只要实现 std::fmt 模块的格式化特型，就可以扩展这些宏以支持自定义类型。此外，使用 format_args! 宏和 std::fmt::Arguments 类型也可以写出自己的支持格式化语言的函数和宏。

格式化宏总是借用对其参数的共享引用，不会取得所有权，也不会修改它们。

模板的 {...} 部分称为**格式化形参**，形式为 {*which*:*how*}。*which* 和 *how* 都是可选的，{} 用的也很多。

格式化形参的 *which* 部分用于选择使用模板后面的哪个参数来填补当前位置。可以通过索引或名称来选择参数。没有 *which* 部分的形参会简单地从左往右应用参数。

格式化形参的 *how* 部分用于指定如何格式化参数：加多少空白、精度如何、基数多少，等等。如果有 *how*，则其前面的冒号是必需的。

表 17-3 是一些例子。

表17-3：格式化值的例子

模板字符串	参数列表	结　　果
"number of {}: {}"	"elephants", 19	"number of elephants: 19"
"from {1} to {0}"	"the grave", "the cradle"	"from the cradle to the grave"
"v = {:?}"	vec![0,1,2,5,12,29]	"v = [0, 1, 2, 5, 12, 29]"

（续）

模板字符串	参数列表	结　果
"name = {:?}"	"Nemo"	"name = \"Nemo\""
"{:8.2} km/s"	11.186	" 11.19 km/s"
"{:20} {:02x} {:02x}"	"adc #42", 105, 42	"adc #42 69 2a"
"{1:02x} {2:02x} {0}"	"adc #42", 105, 42	"69 2a adc #42"
"{lsb:02x} {msb:02x} {insn}"	insn="adc #42", lsb=105, msb=42	"69 2a adc #42"

如果想让输出包含 { 或 } 字符，那么可以在模板中用两个相应的字符：

```
assert_eq!(format!("{{a, c}}    {{a, b, c}}"),
           "{a, c}    {a, b, c}");
```

17.4.1　格式化文本值

在格式化 &str 或 String（char 被看作只有一个字符的字符串）这样的文本类型时，形参的 *how* 部分可以包含几个组件，都是可选的。

- **文本长度限制**。如果参数比这个值长，Rust 会将参数截短。如果没有指定这个值，Rust 则使用完整的文本。
- **最小字段宽度**。截短之后，如果参数比这个值小，Rust 会在右侧（默认）填充空格（默认），从而让字段达到指定宽度。如果省略，Rust 则不会填充参数。
- **对齐**。如果参数需要填充空格以满足最小字段宽度，那么这个值表示文本在字段中应该处于什么位置。<、^ 和 > 分别把文本放在开始、中间和末尾。
- **填充**字符，用于填充过程。如果省略，Rust 会使用空格填充。如果指定了填充字符，也必须同时指定对齐。

表 17-4 是一些格式化文本值的例子及效果。全部示例使用的都是 8 个字符的参数 "bookends"。

表17-4：格式化文本值

使用的特性	模板字符串	结　果
默认	"{}"	"bookends"
最小字段宽度	"{:4}"	"bookends"
	"{:12}"	"bookends "
文本长度限制	"{:.4}"	"book"
	"{:.12}"	"bookends"
字段宽度及长度限制	"{:12.20}"	"bookends "
	"{:4.20}"	"bookends"
	"{:4.6}"	"booken"
	"{:6.4}"	"book "
左对齐，字段宽度	"{:<12}"	"bookends "
居中，字段宽度	"{:^12}"	" bookends "
右对齐，字段宽度	"{:>12}"	" bookends"
填充 =，居中，字段宽度	"{:=^12}"	"==bookends=="
填充 *，右对齐，字段宽度，长度限制	"{:*>12.4}"	"********book"

Rust 的格式化对宽度的理解是朴素的：它假设每个字符占一列，不考虑组合字符、半宽片假名、零宽空格，以及其他与 Unicode 相关的乱七八糟的东西。比如：

```
assert_eq!(format!("{:4}", "th\u{e9}"),   "th\u{e9} ");
assert_eq!(format!("{:4}", "the\u{301}"), "the\u{301}");
```

虽然 Unicode 认为这两个字符串都相当于 "thé"，但 Rust 格式化程序并不知道像 '\u{301}'（COMBINING ACUTE ACCENT，组合重音符）这样的字符需要特殊对待。对第一个字符串，它会正确地填充空格，但对第二个字符串，它认为已经 4 列宽了，因此不会再填充空格。虽然对这个特例而言，好像看起来让 Rust 改进一下也不难，但面向所有 Unicode 脚本的真正的多语言文本格式化是一项艰巨的工作。最好是使用基于平台的用户界面工具，或者干脆生成 HTML 和 CSS 让 Web 浏览器去处理。

除了 &str 和 String 之外，也可以给格式化宏传入引用目标为文本的智能指针类型，比如 Rc<String> 或 Cow<'a, str>。

由于文件名路径不一定是格式良好的 UTF-8，因此 std::path::Path 并非一个文本化类型，不能直接把 std::path::Path 传给格式化宏。不过，Path 的 display 方法返回的值倒是可以按照平台相关的方式格式化：

```
println!("processing file: {}", path.display());
```

17.4.2 格式化数值

当格式化参数具有 usize 或 f64 这样的数值类型时，形参 how 的值包含以下组成部分（全部都是可选的）。

- **填充**及**对齐**。与针对文本类型时的用法一样。
- + 字符表示始终显示数值的符号，即使参数是正值。
- # 字符表示要有明确的基数前缀，比如 0x 或 0b。参见下面的“记数法”项。
- 0 符号表示通过包含前置的 0 来满足最小字段宽度条件（而不是通常的填充方式）。
- **最小字段宽度**。如果格式化后数值没有达到这个宽度，那么 Rust 会在左侧（默认）填充空格（默认），以便达到这个宽度。
- 浮点参数的**精度**。告诉 Rust 应该在小数点后面保留几位数字。Rust 通过舍入或补零来生成必要的小数位。如果省略这个精度参数，那么 Rust 会尽可能用更少的位数来精准表示数值。如果参数是整数，则忽略这个精度。
- **记数法**。对整数类型来说，可以是表示二进制的 b、表示八进制的 o 或表示十六进制的 x 或 X。如果包含了 # 字符，就表示使用 Rust 风格的基数前缀 0b、0o、0x 或 0X。对于浮点数类型来说，基数 e 或 E 表示科学记数法，包括一个规范化的系数和 e 或 E 表示的指数。如果没有指定记数法，那么 Rust 会按照十进制来格式化数值。

表 17-5 是格式化 i32 值 1234 的一些示例。

表17-5：格式化数值

使用的特性	模板字符串	结　　果
默认	`"{}"`	`"1234"`
强制符号	`"{:+}"`	`"+1234"`
最小字段宽度	`"{:12}"`	`"        1234"`
	`"{:2}"`	`"1234"`
符号，宽度	`"{:+12}"`	`"       +1234"`
前置零，宽度	`"{:012}"`	`"000000001234"`
符号，前置零，宽度	`"{:+012}"`	`"+00000001234"`
左对齐，宽度	`"{:<12}"`	`"1234        "`
居中，宽度	`"{:^12}"`	`"    1234    "`
右对齐，宽度	`"{:>12}"`	`"        1234"`
左对齐，符号，宽度	`"{:<+12}"`	`"+1234       "`
居中，符号，宽度	`"{:^+12}"`	`"   +1234    "`
右对齐，符号，宽度	`"{:>+12}"`	`"       +1234"`
填充 =，居中，宽度	`"{:=^12}"`	`"====1234===="`
二进制记数法	`"{:b}"`	`"10011010010"`
宽度，八进制记数法	`"{:12o}"`	`"        2322"`
符号，宽度，十六进制记数法	`"{:+12x}"`	`"        +4d2"`
符号，宽度，大写十六进制记数法	`"{:+12X}"`	`"        +4D2"`
符号，基数前缀，宽度，十六进制	`"{:+#12x}"`	`"      +0x4d2"`
符号，基数，补零，宽度，十六进制	`"{:+#012x}"`	`"+0x0000004d2"`
	`"{:+#06x}"`	`"+0x4d2"`

正如最后两个例子所示，最小字段宽度适用于整个数值、符号、基数前缀，所有一切。

负数始终带符号。结果跟所有“强制符号”的示例差不多。

在指定前置零的情况下，对齐和填充字符会被忽略，因为零会扩展数值以填充整个字段。

使用参数 1234.5678 可以展示浮点数类型的情况，如表 17-6 所示。

表17-6：格式化浮点值

使用的特性	模板字符串	结　　果
默认	`"{}"`	`"1234.5678"`
精度	`"{:.2}"`	`"1234.57"`
	`"{:.6}"`	`"1234.567800"`
最小字段宽度	`"{:12}"`	`"   1234.5678"`
宽度，精度	`"{:12.2}"`	`"     1234.57"`
	`"{:12.6}"`	`" 1234.567800"`
前置零，宽度，精度	`"{:012.6}"`	`"01234.567800"`
科学记数法	`"{:e}"`	`"1.2345678e3"`
科学记数法，精度	`"{:.3e}"`	`"1.235e3"`
科学记数法，宽度，精度	`"{:12.3e}"`	`"     1.235e3"`
	`"{:12.3E}"`	`"     1.235E3"`

17.4.3　格式化其他类型

除了字符串和数值，还可以对标准库中的其他几种类型进行格式化。

- 错误类型全部可以直接格式化，这样就很容易将它们包含在错误消息中。每种错误类型都应该实现 `std::error::Error` 特型，该特型扩展了默认的格式化特型 `std::fmt::Display`。因此，任何实现 `Error` 的类型都是可以格式化的。
- 可以格式化互联网协议地址类型如 `std::net::IpAddr` 和 `std::net::SocketAddr`。
- `true` 和 `false` 这两个布尔值也可以格式化，尽管它们通常不是直接呈现给最终用户的最佳字符串。

格式化上述类型时应该使用与格式化字符串同样的形参。长度限制、字段宽度和对齐都没问题。

17.4.4　为调试格式化值

为了调试和输出日志，可以使用格式为 `{:?}` 的形参格式化任意 Rust 标准库中的公共类型，应该都能输出对程序员有价值的内容。可以通过它来检查向量、切片、元组、散列表、线程等数百种类型。

比如，编写如下代码：

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert("Portland", (45.5237606,-122.6819273));
map.insert("Shanghai", (25.0375167, 121.5637));
println!("{:?}", map);
```

会输出：

```
{"Shanghai": (25.0375167, 121.5637), "Portland": (45.5237606, -122.6819273)}
```

`HashMap` 和 `(f64, f64)` 类型都知道如何格式化自身，无须我们操心。

如果在格式化形参中加上 `#` 字符，Rust 将美化打印输出结果。把前面的代码改成 `println!("{:#?}", map)` 会导致如下结果：

```
{
    "Shanghai": (
        25.0375167,
        121.5637
    ),
    "Portland": (
        45.5237606,
        -122.6819273
    )
}
```

当然，具体的格式不能保证一点不差，Rust 新版本不排除调整的可能。

如前所述，可以使用 `#[derive(Debug)]` 语法让自定义类型支持 `{:?}`：

```
#[derive(Copy, Clone, Debug)]
struct Complex { r: f64, i: f64 }
```

有了这个定义，就可以使用 {:?} 格式打印 Complex 的值了：

```
let third = Complex { r: -0.5, i: f64::sqrt(0.75) };
println!("{:?}", third);
```

输出如下：

```
Complex { r: -0.5, i: 0.8660254037844386 }
```

这个输出对于调试而言足够了，不过要是 {} 可以用更数学的格式把它们打印出来就更好了，比如：-0.5 + 0.8660254037844386i。17.4.8 节将介绍怎么做到这一点。

17.4.5 为调试格式化指针

正常情况下，如果把任何类型的指针（引用、Box 或 Rc）传给一个格式化宏，这个宏都会跟随指针并格式化其引用目标，指针本身并不重要。可是，为了调试有时候确实需要把指针打印出来。可以把这个地址当成对应值的“名字”，而这个名字在检查涉及循环和共享的结构时会很有用。

{:p} 符号用于将引用、Box 和其他类指针类型格式化为地址：

```
use std::rc::Rc;
let original = Rc::new("mazurka".to_string());
let cloned = original.clone();
let impostor = Rc::new("mazurka".to_string());
println!("text:     {}, {}, {}",       original, cloned, impostor);
println!("pointers: {:p}, {:p}, {:p}", original, cloned, impostor);
```

以上代码输出为：

```
text:     mazurka, mazurka, mazurka
pointers: 0x7f99af80e000, 0x7f99af80e000, 0x7f99af80e030
```

当然，特定指针的值每次运行都会不一样。但即便如此，通过比较地址也可以清楚地知道前两个引用是同一个 String，第三个则指向一个不同的值。

地址就是一串十六进制值，不太好记，如果能对人类更友好就更好了。但不管怎样，{:p} 的输出总不失为一种简单快捷的方案。

17.4.6 通过索引或名字引用参数

格式化形参可以显示选择它使用的参数。比如：

```
assert_eq!(format!("{1},{0},{2}", "zeroth", "first", "second"),
           "first,zeroth,second");
```

当然，冒号后面可以再跟其他格式化形参：

```
assert_eq!(format!("{2:#06x},{1:b},{0:=>10}", "first", 10, 100),
           "0x0064,1010,=====first");
```

除了通过索引选择参数，还可以使用名字。这样可以让包含较多参数的复杂模板变得更清晰。比如：

```
assert_eq!(format!("{description:.<25}{quantity:2} @ {price:5.2}",
                   price=3.25,
                   quantity=3,
                   description="Maple Turmeric Latte"),
           "Maple Turmeric Latte..... 3 @ 3.25");
```

（这里的命名参数类似于 Python 中的关键字参数，但这只是格式化宏的一种特殊特性，Rust 函数调用语法并不支持。）

此外，还可以在一个格式化宏中混用索引、名字和位置（既不是索引也不是名字）参数，其中，位置参数从左往右匹配，但不考虑已有的索引和命名参数：

```
assert_eq!(format!("{mode} {2} {} {}",
                   "people", "eater", "purple", mode="flying"),
          "flying purple people eater");
```

混合使用时，命名参数必须放在列表末尾。

17.4.7　动态宽度与精度

形参的最小字段宽度、文本长度限制和数值精度并不一定是固定的值，而是可以在运行时确定。

之前我们看到过像这个表达式这样的例子，它会让字符串 `content` 在 20 字符宽的字段中右对齐：

```
format!("{:>20}", content)
```

如果想在运行时确定字段宽度，可以这样写：

```
format!("{:>1$}", content, get_width())
```

使用 `1$` 作为最小字段宽度是告诉 `format!` 使用第二个参数的值作为宽度。引用的参数必须是 `usize`。还可以通过名字来引用参数：

```
format!("{:>width$}", content, width=get_width())
```

同样，文本长度限制也可以如法炮制：

```
format!("{:>width$.limit$}", content,
        width=get_width(), limit=get_limit())
```

在文本长度限制或浮点精度的位置，还可以用 `*` 表示要取得下一个位置上的参数作为精度。下面的代码会把 `content` 切为最长 `get_limit()` 个字符：

```
format!("{:.*}", get_limit(), content)
```

作为精度的参数必须是 `usize`。字段宽度没有对应的语法。

17.4.8　格式化自定义类型

格式化宏使用 `std::fmt` 模块中定义的一组特型将值转换为文本。只要你自己实现其中一个或多个特型，就可以让 Rust 的格式化宏格式化自定义类型。

格式形参的记号表示其参数类型必须实现哪个特型，如表 17-7 所示。

表17-7：格式化自定义类型

记　号	示　例	特　型	用　途
无	{}	std::fmt::Display	文本、数值、错误：兜底的特型
b	{:#b}	std::fmt::Binary	二进制中的数值
o	{:#5o}	std::fmt::Octal	八进制中的数值
x	{:4x}	std::fmt::LowerHex	十六进制中的数值，小写数字
X	{:016X}	std::fmt::UpperHex	十六进制中的数值，大写数字
e	{:.3e}	std::fmt::LowerExp	科学记数法中的浮点数值
E	{:.3E}	std::fmt::UpperExp	同上，E 大写
?	{:#?}	std::fmt::Debug	调试视图，适合开发者
p	{:p}	std::fmt::Pointer	指针地址，适合开发者

如果把 `#[derive(Debug)]` 属性放在类型定义上，就可以使用 `{:?}` 格式形参。添加这个属性就是让 Rust 帮你实现 `std::fmt::Debug` 特型。

格式化特型的结构都一样，只是名字不同。下面以 `std::fmt::Display` 为例来看一看：

```
trait Display {
    fn fmt(&self, dest: &mut std::fmt::Formatter)
        -> std::fmt::Result;
}
```

这个 `fmt` 方法的任务就是生成 `self` 的适当的格式化表示，并将字符写入 `dest`。除了作为输出流，`dest` 参数也会携带解析格式形参后得到的像对齐和最小字段宽度这样的细节。

比如，本章前面在介绍复数的例子时曾提到要是 `Complex` 值能够以惯常的 `a + bi` 形式输出就好了。下面就是为此而对 `Display` 的一个实现：

```
use std::fmt;

impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let i_sign = if self.i < 0.0 { '-' } else { '+' };
        write!(dest, "{} {} {}i", self.r, i_sign, f64::abs(self.i))
    }
}
```

这里利用了 `Formatter` 本身是一个输出流的事实，因此 `write!` 宏可以帮我们做大部分工作。有了以上实现，就可以这样写了：

```
let one_twenty = Complex { r: -0.5, i: 0.866 };
assert_eq!(format!("{}", one_twenty),
           "-0.5 + 0.866i");
```

```
let two_forty = Complex { r: -0.5, i: -0.866 };
assert_eq!(format!("{}", two_forty),
           "-0.5 - 0.866i");
```

有时候，可能需要以极坐标形式显示复数。比如，要在复平面上从原点到数值画一条线，极坐标形式能给出线的长度，以及其顺时针到 x 轴正方向的角度。格式形参中的 `#` 字符通常用于选择替代的显示形式，`Display` 实现可以将其看作对使用极坐标形式的一个要求：

```
impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let (r, i) = (self.r, self.i);
        if dest.alternate() {
            let abs = f64::sqrt(r * r + i * i);
            let angle = f64::atan2(i, r) / std::f64::consts::PI * 180.0;
            write!(dest, "{} ∠ {}°", abs, angle)
        } else {
            let i_sign = if i < 0.0 { '-' } else { '+' };
            write!(dest, "{} {} {}i", r, i_sign, f64::abs(i))
        }
    }
}
```

下面来使用这个实现：

```
let ninety = Complex { r: 0.0, i: 2.0 };
assert_eq!(format!("{}", ninety),
           "0 + 2i");
assert_eq!(format!("{:#}", ninety),
           "2 ∠ 90°");
```

虽然这个格式化特型的 `fmt` 方法返回 `fmt::Result` 值（典型的模块特定的 `Result` 类型），也应该只在 `Formatter` 的操作上传播失败，就像前面的 `fmt::Display` 实现在调用 `write!` 时那样。你的格式化函数绝不能自己产生错误。这样 `format!` 之类的宏就可以简单地返回一个 `String` 而不是 `Result<String, ...>`，因为给 `String` 追加格式化文本永远不会出错。而且这样也能确保由 `write!` 或 `writeln!` 导致的错误能够反映真正来自底层 I/O 流的问题，而不是格式化问题。

`Formatter` 还有很多其他的有用的方法，包括用于处理映射、列表等结构化数据的方法。这些方法本书没有涉及，要了解详细内容请参考在线文档。

17.4.9 在你的代码中使用格式化语言

使用 Rust 的 `format_args!` 宏和 `std::fmt::Arguments` 类型，可以在自己的代码中编写接收格式化模板和参数的函数和宏。比如，假设你的程序需要在运行的时候记录状态消息，此时你希望利用 Rust 的文本格式化语言生成这些消息。下面就是一个起点：

```
fn logging_enabled() -> bool {
    ...
}

use std::fs::OpenOptions;
use std::io::Write;
```

```
fn write_log_entry(entry: std::fmt::Arguments) {
    if logging_enabled() {
        // 为简单起见，每次只是打开文件
        let mut log_file = OpenOptions::new()
            .append(true)
            .create(true)
            .open("log-file-name")
            .expect("failed to open log file");

        log_file.write_fmt(entry)
            .expect("failed to write to log");
    }
}
```

然后可以这样调用 write_log_entry：

```
write_log_entry(format_args!("Hark! {:?}\n", mysterious_value));
```

编译时，format_args! 宏会解析模板字符串并按照参数类型对其进行检查，并在发现问题时报错。运行时，它会求值参数并构建一个带有所有格式化文本必需信息的 Arguments 值，包括模板的预解析形式，以及对参数值的共享引用。

构建 Arguments 值很省事，只是收集几个指针而已。此时并不会发生格式化操作，只是收集后面格式化时所需的信息。这一点很重要：如果不启用打印日志，那么花在转换数值为十进制、填充值等上面的时间都是浪费的。

File 类型实现了 std::io::Write 特型，其 write_fmt 方法接收 Argument 参数并进行格式化。之后再把结果写入底层流。

调用 write_log_entry 的代码并不好看。此时就可以用上宏了：

```
macro_rules! log { // 宏定义的名字后面不需要加叹号!
    ($format:tt, $($arg:expr),*) => (
        write_log_entry(format_args!($format, $($arg),*))
    )
}
```

第 20 章会详细介绍宏。现在，只要知道它定义了一个新的 log! 宏，而这个宏会把它的参数传给 format_args!，并对得到的结果 Arguments 值再调用 write_log_entry 函数就行了。println!、writeln! 和 format! 等格式化宏的实现也都差不多。

现在就可以这样使用 log! 宏了：

```
log!("O day and night, but this is wondrous strange! {:?}\n",
     mysterious_value);
```

是不是觉得好一点了？

17.5　正则表达式

外部的 regex 包是 Rust 官方的正则表达式库，其提供了常用的搜索和匹配功能。这个库对 Unicode 提供了完善的支持，但也可以搜索字节字符串。虽然不支持其他正则表达式包中的某些特性（比如反向引用和前后看模式），但这些简化允许 regex 确保搜索时间在表达式

大小和被搜索文本的长度上呈线性关系。此外，还可以确保即使用不受信任的表达式搜索不受信任的文本，`regex` 也是安全的。

本书将对 `regex` 做一些简单介绍，详细信息请参考在线文档。

尽管 `regex` 包不在 `std` 里，但它仍然是由负责 `std` 的 Rust 库团队维护的。要使用 `regex`，需要在包的 Cargo.toml 文件的 `[dependencies]` 部分添加一行代码：

```
regex = "1"
```

然后在包的底层加一个 `extern crate` 特性项：

```
extern crate regex;
```

下面几节假设已经完成了这些准备。

17.5.1 基本用法

`Regex` 值表示解析之后的正则表达式，随时可用。`Regex::new` 构造函数会将传入的 `&str` 作为正则表达式来解析，返回 `Result`：

```
use regex::Regex;

// semver版本号，比如0.2.1
// 可能包含预发布版本后缀，比如0.2.1-alpha
//（为简单起见，不考虑构建元数据后缀）
//
// 注意，使用r"..."原始字符串语法可以避免过多反斜杠
let semver = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")?;

// 简单搜索，返回布尔值结果
let haystack = r#"regex = "0.2.5""#;
assert!(semver.is_match(haystack));
```

`Regex::captures` 方法从字符串中搜索第一个匹配，返回的 `regex::Captures` 值包含正则表达式中每一组对应的匹配信息：

```
// 可以读取捕获组：
let captures = semver.captures(haystack)
    .ok_or("semver regex should have matched")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

如果相应的捕获组没有匹配，则访问 `Captures` 值的对应索引会导致诧异。要测试某个特定组是否有匹配结果，可以调用 `Captures::get`，它会返回 `Option<regex::Match>`。`Match` 值记录一个捕获组的匹配信息：

```
assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

可以遍历一个字符串中的所有匹配：

```
let haystack = "In the beginning, there was 1.0.0. \
                For a while, we used 1.0.1-beta, \
                but in the end, we settled on 1.2.4.";

let matches: Vec<&str> = semver.find_iter(haystack)
    .map(|match_| match_.as_str())
    .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

迭代器 `find_iter` 会为表达式的每个不重叠匹配分别生成一个 `Match` 值，从字符串开头到末尾。`captures_iter` 方法也类似，只不过生成的是记录所有捕获组的 `Captures` 值。由于记录捕获组会导致搜索变慢，因此如果不需要捕获组信息，那么最好使用不返回它们的方法。

17.5.2 懒构建Regex值

`Regex::new` 构造函数的开销可能很大：在配置比较高的开发机上为 1200 字符的正则表达式构建 `Regex` 会花 1 毫秒时间，随随便便一个表达式也要占用几微秒。为此，最好不要在大计算量的循环中构建 `Regex`，而是只构建 `Regex` 一次，然后重用它。

`lazy_static` 包提供了首次使用时懒构建静态值的优雅方式。要使用它，记住在 Cargo.toml 文件中加上：

```
[dependencies]
lazy_static = "1.4.0"
```

这个包提供了声明这种变量的宏：

```
#[macro_use]
extern crate lazy_static;

lazy_static! {
    static ref SEMVER: Regex
        = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")
              .expect("error parsing regex");
}
```

这个宏会扩展为名为 `SEMVER` 的一个静态变量声明，不过其类型并不是 `Regex`。这个变量的类型是由宏生成的并且实现了 `Deref<Target=Regex>`，因此暴露了与 `Regex` 完全相同的方法。首次解引用 `SEMVER` 时，会运行初始化程序，然后得到的值被保存起来供后面使用。因为 `SEMVER` 是一个静态变量，而不是局部变量，所以每个程序执行最多运行一次初始化程序。

有了上面的声明，使用 `SEMVER` 也很简单：

```
use std::io::BufRead;

let stdin = std::io::stdin();
for line in stdin.lock().lines() {
    let line = line?;
    if let Some(match_) = SEMVER.find(&line) {
        println!("{}", match_.as_str());
    }
}
```

可以把 lazy_static! 声明放到一个模块中，甚至放到一个使用 Regex 的函数内部，只要作用域合适就没问题。无论放在哪里，正则表达式都只会在程序运行时编译一次。

17.6 规范化

对于“茶”的法语 thé，很多人会认为它包含 3 个字符。然而，Unicode 实际上有两种方式表示这个词。

- 在**组合**（composed）形式中，thé 包含 3 个字符：'t'、'h' 和 'é'，其中 'é' 是一个码点为 0xe9 的 Unicode 字符。
- 在**分解**（decomposed）形式中，thé 包含 4 个字符：'t'、'h'、'e' 和 '\u{301}'，其中 'e' 是 ASCII 字符，没有重音符，而码点 0x301 是“组合重音符”（COMBINING ACUTE ACCENT），它会给前面的任意字符加上重音符号。

对 Unicode 而言，é 的组合形式或分解形式并无对错之分，相反它们是等价的，因为表示的是同一个抽象字符。Unicode 要求两种形式都应该以相同的方式显示，输入法则被允许生成任意形式，结果用户大都不知道自己会看到或者应该输入哪种形式。（Rust 支持在字符串字面量中直接使用 Unicode 字符，因此如果你不在乎编码，那么可以直接使用 "thé"。为了清晰起见，此处将使用 \u 转义。）

然而，作为 Rust 的 &str 或 String 值，"th\u{e9}" 和 "the\u{301}" 根本不是一回事。它们有不同的长度，比较起来不相等，有不同的散列值，并且顺序也与其他字符串不同：

```
assert!("th\u{e9}" != "the\u{301}");
assert!("th\u{e9}" >  "the\u{301}");

// 散列程序需要积累一系列值的散列，因此计算1的散列会显得很笨重
use std::hash::{Hash, Hasher};
use std::collections::hash_map::DefaultHasher;
fn hash<T: ?Sized + Hash>(t: &T) -> u64 {
    let mut s = DefaultHasher::new();
    t.hash(&mut s);
    s.finish()
}

// 在将来的Rust版本中，这些值可能会不一样
assert_eq!(hash("th\u{e9}"),   0x53e2d0734eb1dff3);
assert_eq!(hash("the\u{301}"), 0x90d837f0a0928144);
```

明确一下，如果你想比较用户提供的文本，或者将其用作散列表或 B 树中的键，那应该先把每个字符串转换成某种规范的形式。

好在 Unicode 规定了字符串的**规范化**形式。如果两个字符串按照 Unicode 的规则应该判定为等价，那么它们规范化的形式应该每个字符都相同。在以 UTF-8 编码的情况下，就是每个字节都相同。这意味着可以使用 == 来比较规范化的字符串、将它们作为 HashMap 或 HashSet 的键，等等。如此就能享受 Unicode 的平等了。

不进行规范化有时候会导致安全隐患。比如，如果你的网站对用户名在有些情况下进行了规范化，而在另一些情况下没有进行规范化，那么就可能出现两个名为 bananasflambé 的

不同用户。对这两个用户，你的一些代码会认为是一个人，而另一些代码会认为是两个人，最终导致用户利益受损。当然，有很多方法能够避免这种问题，但历史表明，也有很多方法不能避免。

17.6.1 规范化形式

Unicode 定义了 4 种规范化形式，每一种都适用于不同用途。为此，需要回答两个问题。

- 第一，你倾向于字符尽可能**组合**，还是**分解**？

  比如，越南语 Phở 最常用的组合表示形式是 3 个字符的字符串 `"Ph\u{1def}"`，这种情况下，音调记号 ̉ 和元音记号 ̛ 在一个 Unicode 字符上都应用给了基本字符 `o`，即 `'\u{1edf}'`，这个字符的 Unicode 名字叫“LATIN SMALL LETTER O WITH HORN AND HOOK ABOVE”（上带角号和钩号的小写拉丁字母 O）。

  最常用的分解表示形式就是把基本字符跟它的两个记号分开表示成 3 个独立的 Unicode 字符：`'o'`、`'\u{31b}'`（COMBINING HORN，组合角号）和 `'\u{309}'`（COMBINING HOOK ABOVE，组合上钩号）。最终结果是 `"Pho\u{31b}\u{309}"`。（任何时候，只要组合记号以独立字符出现，而不是作为组合字符的一部分，所有规范化形式都会规定它们出现的先后次序。因此就算字符有多个发音记号，其规范化形式都是有完善定义的。）

  组合形式的兼容性问题通常较少，因为它最接近大多数语言在 Unicode 之前就有的自然表现形式。而且，可能也更适合 Rust 的 `format!` 宏之类的简单字符串格式化特性。另一方面，分解形式则可能更适合显示文本或搜索，因为它能呈现出文本的所有细节。

- 第二，如果两个字符序列表示相同的基础文本，但在该文本格式化上有差异，那么你倾向于认为它们等价，还是不同？

  Unicode 对数字 `'5'` 及其上标形式 `'⁵'`（或 `'\u{2075}'`）、圆圈形式 `'⑤'`（或 `'\u{2464}'`）都有单独的字符，但声明这 3 个字符“兼容性等效”（compatibility equivalent）。类似地，Unicode 专门有一个字符表示 ﬃ 的连字形式（`'\u{fb03}'`），但又声明这个字符与 3 个字符的序列 `"ffi"` 兼容性等效。

  兼容性等效对搜索而言是有意义的。比如，搜索只包含 ASCII 字符的 `"difficult"`，也应该匹配使用 ﬃ 连字的字符串 `"di\u{fb03}cult"`。基于兼容性对后一字符串进行分解，把连字替换成 3 个字母 `"ffi"`，可以让搜索变得更容易。但将文本规范化为兼容性等效形式有可能丢失重要信息，因此不可鲁莽行事。比如，多数情况下把 `"2⁵"` 保存为 `"25"` 是错误的。

Unicode NFC（Normalization Form C，规范化形式 C）和 NFD（Normalization Form D，规范化形式 D）对每个字符分别应用最大化组合和最大化分解的策略，但不会尝试统一兼容性等效序列。NFKC 和 NFKD 规范化形式与 NFC 和 NFD 类似，但会将所有兼容性等效序列规范化为各自的某种简单表示。

W3C（World Wide Web Consortium，万维网联盟）的“Character Model For the World Wide Web”推荐对所有内容使用 NFC。Unicode 的“Identifier and Pattern Syntax”附录建议对编程语言的标识符使用 NFKC，并给出了必要时修改该形式的规则。

17.6.2 unicode-normalization包

Rust 的 unicode-normalization 包提供了一个特型，该特型可以给 &str 添加把文本转换为任意 4 种规范化形式的方法。要使用这个包，先在 Cargo.toml 文件的 [dependencies] 部分加上下面这行代码：

```
unicode-normalization = "0.1.8"
```

包的顶部文件还要添加 extern crate 声明：

```
extern crate unicode_normalization;
```

有了上面的声明之后，&str 就拥有了 4 个新方法，它们都返回字符串的对应规范化形式的迭代器：

```
use unicode_normalization::UnicodeNormalization;

// 无论左边的字符串是哪种表示形式（不能只凭眼睛看），这些断言都成立
// these assertions will hold.
assert_eq!("Phở".nfd().collect::<String>(), "Pho\u{31b}\u{309}");
assert_eq!("Phở".nfc().collect::<String>(), "Ph\u{1edf}");
// 这里左边使用的是"ffi"连字符
assert_eq!("① Di\u{fb03}culty".nfkc().collect::<String>(), "1 Difficulty");
```

把规范化之后的字符串再次进行相同形式的规范化会得到相同的文本。

虽然规范化字符串的任意子字符串本身是规范化的，但两个规范化的字符串拼接起来未必是规范化的。比如，第二个字符串以组合字符开头，而对于第一个字符串而言，这个组合字符应该放在它的最后一个组合字符前面。

Unicode 保证，只要文本在规范化时没有使用未分配的码点，那其规范化形式在标准的未来版本中就不会变。这意味着规范化形式通常适合持久存储，而不会受 Unicode 标准演进的影响。

第 18 章

输入和输出

杜利特尔：你有什么证据能证明你的存在？

炸弹 20 号：嗯……好吧……我思，故我在。

杜利特尔：很好，非常好。但是你又如何知道其他东西是存在的呢？

炸弹 20 号：我的传感器传达给我的。

——奇幻动画《黑星》（*Dark Star*）

Rust 标准库针对输入和输出的特性是通过 3 个特型，即 `Read`、`BufRead` 和 `Write`，以及实现它们的各种类型暴露出来的。

- 实现 `Read` 的值有读取字节输入的方法，这些值叫**读取器**（reader）。
- 实现 `BufRead` 的值是**缓冲**读取器，其支持 `Read` 的所有方法，额外又支持读取文本行等的方法。
- 实现 `Write` 的值既支持字节输出也支持 UTF-8 文本输出，这些值叫**写入器**（writer）。

图 18-1 展示了这 3 个特型以及几个读取器和写入器类型的示例。

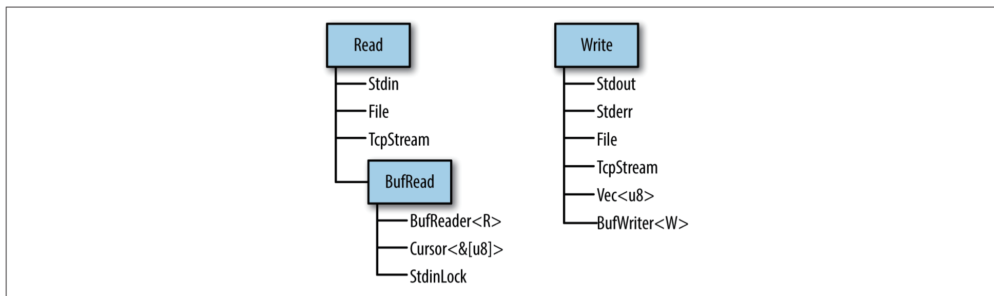


图 18-1：Rust 标准库中的部分读取器和写入器类型

本章将介绍如何使用这些特型和它们的方法、实现它们的类型，以及其他与文件、终端和网络交互的方式。

18.1　读取器和写入器

读取器是一种值，我们的程序可以通过它读取字节，主要有以下例子。

- 使用 `std::fs::File::open(filename)` 打开的文件。
- 用于从网络接收数据的 `std::net::TcpStream`。
- 用于从进程的标准输入流读取数据的 `std::io::stdin()`。
- `std::io::Cursor<&[u8]>` 值，这是一种从内存的字节数组中“读取”数据的读取器。

写入器也是一种值，我们的程序可以通过它写入字节，主要有以下例子。

- 使用 `std::fs::File::create(filename)` 打开的文件。
- 用于通过网络发送数据的 `std::net::TcpStream`。
- 用于将数据写入终端的 `std::io::stdout()` 和 `std::io:stderr()`。
- `std::io::Cursor<&mut [u8]>` 值，此值允许将任何可修改字节切片作为文件写入。
- `Vec<u8>` 也是一个写入器，其 `write` 方法可以为向量追加元素。

因为读取器和写入器都有标准的特型（`std::io::Read` 和 `std::io::Write`），所以常见的做法是编写泛型代码，以涵盖各种输入和输出渠道。例如，下面这个函数可以从任何读取器将全部字节复制到任何写入器：

```
use std::io::{self, Read, Write, ErrorKind};

const DEFAULT_BUF_SIZE: usize = 8 * 1024;

pub fn copy<R: ?Sized, W: ?Sized>(reader: &mut R, writer: &mut W)
    -> io::Result<u64>
    where R: Read, W: Write
{
    let mut buf = [0; DEFAULT_BUF_SIZE];
    let mut written = 0;
    loop {
        let len = match reader.read(&mut buf) {
            Ok(0) => return Ok(written),
            Ok(len) => len,
            Err(ref e) if e.kind() == ErrorKind::Interrupted => continue,
            Err(e) => return Err(e),
        };
        writer.write_all(&buf[..len])?;
        written += len as u64;
    }
}
```

这是 Rust 标准库中 `std::io::copy()` 的实现。因为是泛型的，所以可以使用它将数据从 `File` 复制到 `TcpStream`，从 `Stdin` 复制到内存中的 `Vec<u8>`，等等。

如果对这里的错误处理代码不清楚，可以回顾一下第 7 章。后面会经常使用 Result，掌握这个类型非常重要。

4 个常用的 std::io 的特型 Read、BufRead、Write 和 Seek 有一个专门的前置模块只包含它们：

```
use std::io::prelude::*;
```

这行代码在本章中还会出现一两次。我们也会习惯性地导入 std::io 模块自身：

```
use std::io::{self, Read, Write, ErrorKind};
```

关键字 self 在这里将 io 声明为 std::io 模块的别名。这样，std::io::Result 和 std::io::Error 就可以写成更简单的 io::Result 和 io::Error 了。

18.1.1 读取器

std::io::Read 有几个读取数据的方法，这些方法都以读取器本身的 mut 引用为参数。

- **reader.read(&mut buffer)** 从数据源读取某些字节，然后存储到给定的 buffer 中。buffer 参数的类型是 &mut [u8]。这个方法会读取 buffer.len() 字节。
  返回类型是 io::Result<u64>，这是 Result<u64, io::Error> 类型别名。操作成功，u64 值是读取的字节数，可能等于或小于 buffer.len()，即使数据源还有数据也不会更多，这要看数据源。Ok(0) 意味着没有输入要读取了。

  操作错误，.read() 返回 Err(err)，其中 err 是一个 io::Error 值。io::Error 值是可以打印的，其格式适合人类阅读。对于程序，它有一个 .kind() 方法，此方法返回 io::ErrorKind 类型的错误码。这个枚举的成员中有 PermissionDenied 和 ConnectionReset 等。多数表示不能忽略的严重错误，需要特殊处理。io::ErrorKind::Interrupted 对应 Unix 错误码 EINTR，意思是读取操作被某个信号中断了。除非程序有处理信号的逻辑，否则这时候就应该重试。前一节中 copy() 的代码就给出了这种情况下重新读取的例子。

  如你所见，.read() 方法是非常低级的，甚至还继承了底层操作系统的怪癖。如果给数据源的新类型实现 Read 特型，这会有很多余地。如果是读取某些数据，则会很痛苦。因此，Rust 提供了一些高级的便捷方法。这些方法默认都实现了 .read() 的逻辑，而且也都会自动处理 ErrorKind::Interrupted，所以你就不用自己处理了。

- **reader.read_to_end(&mut byte_vec)** 从读取器中读出所有剩余输入，追加到 byte_vec 中。byte_vec 是一个 Vec<u8>。这个方法返回 io::Result<(usize)>，即读到的字节数。
  这个方法对添加到向量中的数据量没有限制，因此不要对不可信的数据源使用它。（可以使用 .take() 方法来添加限制，如下所述。）

- **reader.read_to_string(&mut string)** 类似，只是将数据追加到给定的 String。如果输入流不是有效的 UTF-8，这个方法会返回 ErrorKind::InvalidData 错误。

  在某些语言中，字节输入和字符输入由不同类型处理。目前，UTF-8 的地位如日中天，Rust 认可这个事实标准并在所有地方都支持 UTF-8。其他字符集通过开源的 encoding 包支持。

- **`reader.read_exact(&mut buf)`** 读取恰好足够的数据填充到给定的 buffer 中。这个参数类型是 &[u8]。如果读取器在读到 buf.len() 字节前已经把数据读完了，这个方法会返回 ErrorKind::UnexpectedEof 错误。

以上都是 Read 特型的主要方法。此外，还有 4 个适配器方法，以 reader 的值为参数，将其转换为一个迭代器或一个不同的读取器。

- **`reader.bytes()`** 返回输入流字节的迭代器。迭代器项的类型是 io::Result<u8>，因此每个字节都需要错误检查。另外，这个方法对每个字节都会调用一次 reader.read()，这对没有缓冲的读取器来说效率很低。
- **`reader.chars()`** 类似，只不过迭代器项是字符，将输入当作 UTF-8。无效的 UTF-8 导致 InvalidData 错误。
- **`reader.chain(reader2)`** 返回新读取器，产生 reader 的所有输入和 reader2 的所有输入。
- **`reader.take(n)`** 返回新读取器，从与 reader 相同的数据源读取输入，但只读取 *n* 字节。

没有关闭读取器的方法。读取器和写入器通常都会实现 Drop，因而会自动关闭。

18.1.2　缓冲读取器

为提高效率，读取器和写入器可以缓冲起来。所谓**缓冲**，简单来说就是给读取器和写入器一块内存（缓冲区），用来暂时保存输入和输出数据。缓冲可以减少系统调用，如图 18-2 所示。在这个例子中，应用通过调用 .read_line() 方法从 BufReader 中读取数据。而 BufReader 再从操作系统上读取更大的数据块。

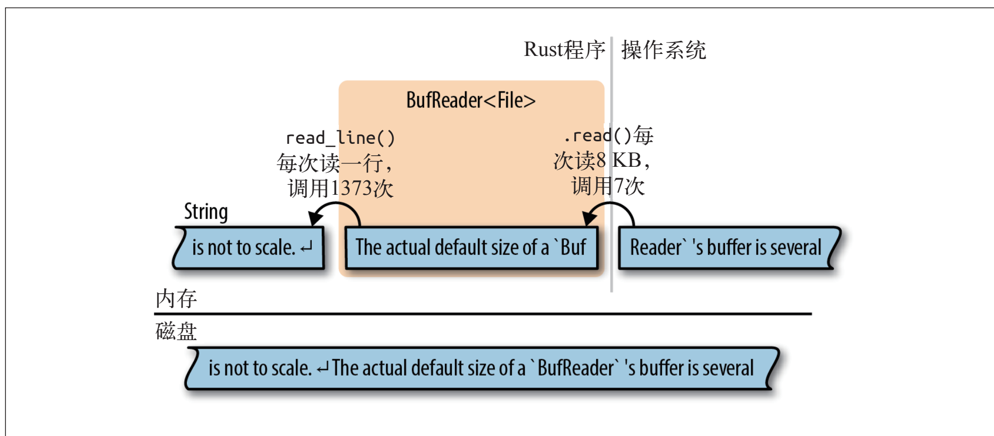


图 18-2：缓冲文件读取器示例

这张图是不成比例的。BufReader 缓冲区的实际默认大小有几千字节，因此一次系统 read 可以应付几百次 .read_line() 调用。因为系统调用相对较慢，所以这就显得很重要的了。

（如图所示，操作系统也有一个缓冲区，原因相同：系统调用慢，但从磁盘读取数据更慢。）

缓冲读取器实现了 Read 和 BufRead 两个特型，后者增加了以下方法。

- **reader.read_line(&mut line)** 读取一行文本并追加到 line，line 是一个 String。行尾的换行符 '\n' 也会包含在 line 中。如果输入有 Windows 风格的行终结符 "\r\n"，则这两个字符也会包含在 line 中。

  返回值是 io::Result<usize>，表示读取的字节数，包括行终结符（如果有的话）。

  如果读取器处于输入末尾，则 line 不变且返回 Ok(0)。

- **reader.lines()** 返回输入行的迭代器。迭代项类型是 io::Result<String>。换行符**不会**包含在字符串中。如果输入中有 Windows 风格的行终结符 "\r\n"，则这两个字符都会被去掉。

  这个方法基本上可以满足读取文本输入的需求。接下来的两节将展示使用它的几个例子。

- **reader.read_until(stop_byte, &mut byte_vec)** 和 **reader.split(stop_byte)** 与 .read_line() 和 .lines() 类似，只是以字节为单位，产生 Vec<u8> 而非 String。stop_byte 表示定界符。

BufRead 还提供了两个低级设计方法：.fill_buf() 和 .consume(n)，用于直接访问读取器内部的缓冲区。关于这两个方法的详细信息，请参考在线文档。

接下来的两节将详细介绍缓冲读取器。

18.1.3 读取文本行

下面这个函数实现了 Unix 的 grep 命令，它会搜索多行文本，通常与其他命令通过管道组合使用，以查找指定写入器：

```
use std::io;
use std::io::prelude::*;

fn grep(target: &str) -> io::Result<()> {
    let stdin = io::stdin();
    for line_result in stdin.lock().lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

因为我们想调用 .lines()，所以需要输入源实现 BufRead。此时，调用 io::stdin() 取得输入的数据。不过 Rust 标准库用互斥量保护 stdin。为此要调用 .lock() 给 stdin 加锁以明确当前线程专有使用。这个方法返回实现 BufRead 的 StdinLock 值。在循环最后，StdinLock 会被清除，释放互斥量。（没有互斥量，两个线程同时读取 stdin 会导致未定义行为。C 语言也有相同问题，且以使用方式来解决。所有 C 标准库输入和输出函数都会在后台得到一把锁。唯一的区别是在 Rust 中需要显式获得锁。）

这个函数的其他地方都很简单：调用 .lines() 之后遍历得到的迭代器。因为这个迭代器产生 Result 值，所以使用 ? 操作符检查错误。

假设我们想进一步扩展这个 greg 程序，增加搜索磁盘上文件的功能。可以把它改写为泛型函数：

```
fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

这样就既可以给它传 StdinLock 也可以传缓冲 File 了：

```
let stdin = io::stdin();
grep(&target, stdin.lock())?; // 没问题

let f = File::open(file)?;
grep(&target, BufReader::new(f))?; // 同样没问题
```

注意，File 不会自动缓冲，因为它实现的是 Read 而非 BufRead。不过，为 File 或其他非缓冲读取器创建缓冲读取器很容易。BufReader::new(reader) 就可以实现。（设置缓冲区的大小，可以使用 BufReader::with_capacity(size, reader)。）

在大多数语言中，文件默认会被缓冲。如果想要非缓冲的输入或输出，则必须知道如何关闭缓冲。而在 Rust 中，File 和 BufReader 是两个不同的库特性，因为有时候需要不带缓冲的文件，有时候需要非文件的缓冲（例如，缓冲网络输入）。

下面是一个包含错误处理和基本参数解析的完整程序。

```
// grep：搜索stdin或某些文件中匹配指定字符串的行

use std::error::Error;
use std::io::{self, BufReader};
use std::io::prelude::*;
use std::fs::File;
use std::path::PathBuf;

fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}

fn grep_main() -> Result<(), Box<Error>> {
```

```
    // 取得命令行参数。第一个参数是要搜索的字符串，其余参数是文件名
    let mut args = std::env::args().skip(1);
    let target = match args.next() {
        Some(s) => s,
        None => Err("usage: grep PATTERN FILE...")?
    };
    let files: Vec<PathBuf> = args.map(PathBuf::from).collect();

    if files.is_empty() {
        let stdin = io::stdin();
        grep(&target, stdin.lock())?;
    } else {
        for file in files {
            let f = File::open(file)?;
            grep(&target, BufReader::new(f))?;
        }
    }

    Ok(())
}

fn main() {
    let result = grep_main();
    if let Err(err) = result {
        let _ = writeln!(io::stderr(), "{}", err);
    }
}
```

18.1.4　收集行

有些读取器方法（包括 `.lines()`）会返回产生 `Result` 值的迭代器。第一次把文本中的所有行收集到一个大向量中时，你会碰到去除 `Result` 的问题。

```
 // 没问题，但不是你想要的
let results: Vec<io::Result<String>> = reader.lines().collect();

// 错误：不能将Result集合转换为Vec<String>
let lines: Vec<String> = reader.lines().collect();
```

第二次编译不通过。怎么处理这个错误？最直观的方式是编写 `for` 循环，检查每一项是否包含错误：

```
let mut lines = vec![];
for line_result in reader.lines() {
    lines.push(line_result?);
}
```

不错，但要是能在这里使用 `.collect()` 就好了。实际上确实也可以。为此只需弄清楚要取得哪种类型：

```
let lines = reader.lines().collect::<io::Result<Vec<String>>>()?;
```

为什么这样能行？标准库中为 `Result` 实现了 `FromIterator`（查看在线文档很容易略过），

就是这个实现让使用 .collect() 成为可能：

```
impl<T, E, C> FromIterator<Result<T, E>> for Result<C, E>
    where C: FromIterator<T>
{
    ...
}
```

这个实现意味着，如果可以将类型 T 的项收集到类型 C（where C: FromIterator<T>）的集合中，那么就可以将类型 Result<T, E> 的项收集为类型 Result<C, E>（FromIterator<Result<T, E>> for Result<C, E>）。

换句话说，io::Result<Vec<String>> 是一个集合类型，因此 .collect() 方法可以创建并填充该类型的值。

18.1.5　写入器

如前所见，操作输入主要是使用方法完成的。输出则有点不同。

本书中会一直使用 println!() 产生纯文本输出。

```
println!("Hello, world!");

println!("The greatest common divisor of {:?} is {}",
         numbers, d);
```

还有一个不会在行尾添加换行符的 print!() 宏。print!() 和 println!() 的格式化代码与 format! 宏一样，17.4 节曾详细介绍过。

要向一个写入器发送输出，可以使用 write!() 和 writeln!() 宏。它们跟 print!() 和 println!() 类似，只是有两点不同。

```
writeln!(io::stderr(), "error: world not helloable")?;

writeln!(&mut byte_vec, "The greatest common divisor of {:?} is {}",
         numbers, d)?;
```

一个不同是这两个 write 宏都多了一个参数，即第一个参数，这是一个写入器。另一个不同是它们都返回 Result，因此必须处理错误。这也是每行末尾都使用 ? 操作符的原因。

print 宏不返回 Result，它们只会在写入失败时诧异。因为写入的是终端，所以很少失败。

Write 特型有如下方法。

- **writer.write(&buf)** 将切片 buf 中的某些字节写入底层流。这个方法返回 io::Result<usize>，成功则包含写入的字节数，可能小于 buf.len()，这取决于流。
  与 Reader::read() 类似，这是一个低级方法，应该尽量不要直接使用。
- **writer.write_all(&buf)** 将切片 buf 中的所有字节都写入，返回 Result<()>。
- **writer.flush()** 将所有缓冲数据都写到底层流，返回 Result<()>。

与读取器类似，写入器也会在被清除时自动关闭。

就像 `BufReader::new(reader)` 会给任何读取器添加缓冲一样，`BufWriter::new(writer)` 也会给任何写入器添加缓冲。

```
let file = File::create("tmp.txt")?;
let writer = BufWriter::new(file);
```

要设置缓冲区的大小，可以使用 `BufWriter::with_capacity(size, writer)`。

在 `BufWriter` 被清除时，所有剩余缓冲数据都会写入底层写入器。不过，如果在写入期间发生错误，错误则会被**忽略**。（因为错误会在 `BufWriter` 的 `.drop()` 方法中发生，所以没办法报告。）为确保应用可以发现所有输出错误，应该在清除之前，手工使用 `.flush()` 清理缓冲写入器。

18.1.6　文件

本书已经介绍了两种打开文件的方式。

- **`File::open(filename)`** 打开已有文件供读取。这个方法返回一个 `io::Result<File>`，如果文件不存在就是一个错误。
- **`File::create(filename)`** 创建新文件供写入。如果指定名字的文件已存在，则该文件会被删节。

注意 `File` 类型在文件系统模块 `std::fs` 中，不在 `std::io` 中。

如果这两种方式都不能满足要求，则可以使用 `OpenOptions` 指定想要的行为：

```
use std::fs::OpenOptions;

let log = OpenOptions::new()
    .append(true) // 如果文件存在，则在末尾追加内容
    .open("server.log")?;

let file = OpenOptions::new()
    .write(true)
    .create_new(true) // 如果文件存在则失败
    .open("new_file.txt")?;
```

方法 `.append()`、`.write()`、`.create_new()` 等都可以连缀调用，因为它们都返回 `self`。这种方法连缀调用的模式在 Rust 中很常见，它们有一个名字叫**构建器**（builder）。`std::process::Command` 也是一个例子。要了解关于 `OpenOptions` 的更多信息，请参考在线文档。

`File` 打开后，就跟任何其他读取器或写入器一样。可以根据需要添加缓冲。`File` 也会在被清除时自动关闭。

18.1.7　搜寻

`File` 也实现了 `Seek` 特型，这意味着可以在 `File` 里跳转，而不是只能从头到尾一次性读取或写入。`Seek` 的定义如下：

```
pub trait Seek {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64>;
}

pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64)
}
```

由于这个枚举，seek 方法表达力非常强。file.seek(SeekFrom::Start(0)) 表示跳到开始位置，file.seek(SeekFrom::Current(-8)) 表示后退 8 字节。

搜寻文件很慢。无论是机械硬盘还是 SSD（Solid-State Drive，固态硬盘），一次搜寻都相当于读取几兆数据。

18.1.8 其他读取器和写入器类型

本章前面展示了一些实现 Read 和 Write 的非 File 类型的例子。下面再详细介绍一下这些类型。

- **io::stdin()** 返回标准输入流的读取器，类型为 io::Stdin。因为它由所有线程共享，所以每次读取都涉及获得和释放互斥量。

  Stdin 有一个 .lock() 方法用于获得互斥量并返回一个 io::StdinLock，这是一个缓冲读取器，在被清除之前会持有互斥量。因此 StdinLock 上的个别操作可以避免互斥量开销。18.1.3 节展示过使用这个方法的示例代码。

  出于技术原因，io::stdin().lock() 行不通。这个锁保存对 Stdin 值的引用，而这意味着 Stdin 值必须保存在某个生命期足够长的地方：

  ```
  let stdin = io::stdin();
  let lines = stdin.lock().lines(); // 没问题
  ```

- **io::stdout()** 和 **io::stderr()** 返回标准输出和标准错误流的写入器。它们都有互斥量和 .lock() 方法。
- **Vec<u8>** 实现了 Write。写入 Vec<u8> 可以用新数据扩展向量。

  （不过，String **没有**实现 Write。要使用 Write 构建字符串，首先要写到一个 Vec<u8> 中，然后再使用 String::from_utf8(vec) 把向量转换为字符串。）

- **Cursor::new(buf)** 创建了一个新 Cursor，是一个从 buf 中读取数据的缓冲读取器。用于创建读取 String 的读取器。参数 buf 可以是实现 AsRef<[u8]> 的任何类型，因此也可以传入 &[u8]、&str 或 Vec<u8>。

  Cursor 内部很简单，只有两个字段：buf 本身和一个整数，此整数表示在 buf 中的偏移量，也就是下次读取的起点。初始位置是 0。

  Cursor 实现了 Read、BufRead 和 Seek。如果 buf 的类型是 &mut [u8] 或 Vec<u8>，则 Cursor 也实现了 Write。写入游标会从当前位置开始覆盖 buf 中相应的字节。如果写

入字节超出 `&mut [u8]`，则只能写入部分内容或者导致 `io::Error`。不过，使用游标写入超出 `Vec<u8>` 的字节没有问题，向量会自动增长。为此，`Cursor<&mut [u8]>` 和 `Cursor<Vec<u8>>` 实现了 `std::io::prelude` 的所有 4 个特型。

- **`std::net::TcpStream`** 表示 TCP 网络连接。因为 TCP 启用了双向通信，所以它既是读取器又是写入器。

  静态方法 `TcpStream::connect(("hostname", PORT))` 尝试连接服务器，返回 `io::Result<TcpStream>`。

- **`std::process::Command`** 支持创建一个子进程，将数据导入其标准输入，类似下面这样：

```
use std::process::{Command, Stdio};

let mut child =
    Command::new("grep")
    .arg("-e")
    .arg("a.*e.*i.*o.*u")
    .stdin(Stdio::piped())
    .spawn()?;

let mut to_child = child.stdin.take().unwrap();
for word in my_words {
    writeln!(to_child, "{}", word)?;
}
drop(to_child); // 关闭grep的stdin，因此会退出
child.wait()?;
```

`child.stdin` 的类型是 `Option<std::process::ChildStdin>`。这里在设置子进程时使用了 `.stdin(Stdio::piped())`，因此在 `.spawn()` 成功后 `child.stdin` 会被填充数据。如果没有数据，则 `child.stdin` 就是 `None`。

`Command` 也有类似的方法 `.stdout()` 和 `.stderr()`，可以用来在 `child.stdout` 和 `child.stderr` 中请求读取器。

`std::io` 模块也提供了一些函数，返回简单的读取器和写入器。

- **`io::sink()`** 是一个无操作写入器。所有写入方法都返回 `Ok`，但数据会被丢弃。
- **`io::empty()`** 是一个无操作读取器。读取始终成功，但返回输入终止。
- **`io::repeat(byte)`** 返回的读取器会反复给出指定字节。

18.1.9　二进制数据、压缩与序列化

很多开源包在 `std::io` 基础上提供了更多功能。

比如，`byteorder` 包提供了 `ReadBytesExt` 和 `WriteBytesExt` 特型，为所有二进制输入和输出的读取器和写入器提供了方法：

```
use byteorder::{ReadBytesExt, WriteBytesExt, LittleEndian};

let n = reader.read_u32::<LittleEndian>()?;
writer.write_i64::<LittleEndian>(n as i64)?;
```

flate2 包为读、写 gzip 压缩的数据提供了额外的适配器方法：

```
use flate2::FlateReadExt;

let file = File::open("access.log.gz")?;
let mut gzip_reader = file.gz_decode()?;
```

serde 包面向序列化和反序列化，可以实现 Rust 数据结构与字节之间的互相转换。11.2.2 节曾提到过这个包，接下来再详细了解一下。

假设我们有一些文本冒险游戏的映射数据保存在一个 HashMap 中：

```
type RoomId = String;                       // 每个房间有一个唯一的名字
type RoomExits = Vec<(char, RoomId)>;       // ……还有一组出口
type RoomMap = HashMap<RoomId, RoomExits>;  // 房间名到出口的映射，简单

// 创建一个简单的映射
let mut map = RoomMap::new();
map.insert("Cobble Crawl".to_string(),
           vec![('W', "Debris Room".to_string())]);
map.insert("Debris Room".to_string(),
           vec![('E', "Cobble Crawl".to_string()),
                ('W', "Sloping Canyon".to_string())]);
...
```

把这样的数据转换为 JSON 输出只需要几行代码：

```
use std::io;
use serde::Serialize;
use serde_json::Serializer;

let mut serializer = Serializer::new(io::stdout());
map.serialize(&mut serializer)?;
```

以上代码使用了 serde::Serialize 特型的 serialize 方法。这个库为所有知道如何序列化的类型添加了这个特型，也包括我们数据中用到的类型：字符串、字符、元组、向量和 HashMap。

serde 非常灵活。在这个程序中，输出是 JSON 数据，因为我们选择的是 serde_json 序列化器。而 MessagePack 等其他格式也是支持的。类似地，可以把上面的输出发送给一个文件、一个 Vec<u8> 或其他任何写入器。上面的代码会将数据打印到 stdout，示例如下：

```
{"Debris Room":[["E","Cobble Crawl"],["W","Sloping Canyon"]],"Cobble Crawl":
[["W","Debris Room"]]}
```

serde 也包含对派生两个关键 serde 特型的支持：

```
#[derive(Serialize, Deserialize)]
struct Player {
    location: String,
    items: Vec<String>,
    health: u32
}
```

到 Rust 1.17 为止，这个 `#[derive]` 属性要求在项目中多增加几步设置工作。这里就不再介绍了，详细内容可以参考 `serde` 的文档。简单来说，构建系统会自动为 `Player` 实现 `serde::Serialize` 和 `serde::Deserialize`，因此序列化 `Player` 的值就简单了：

```
player.serialize(&mut serializer)?;
```

输出类似下面这样：

```
{"location":"Cobble Crawl","items":["a wand"],"health":3}
```

18.2 文件与目录

接下来的几节将介绍 Rust 操作文件和目标的特性，它们都包含在 `std::path` 和 `std::fs` 模块中。所有这些特性都涉及文件名，因此下面先从文件名相关的类型开始。

18.2.1 OsStr和Path

遗憾的是，操作系统不会强制文件名必须是有效的 Unicode。以下是两个创建文本文件的 Linux 终端命令。只有第一个使用了有效的 UTF-8 文件名。

```
$ echo "hello world" > ô.txt
$ echo "O brave new world, that has such filenames in't" > $'\xf4'.txt
```

两个命令必然都会执行，因为 Linux 内核并不知道 Ogg Vorbis 的 UTF-8。对内核而言，任何字节字符串（不包括空字节和斜杠）都是可以接受的文件名。在 Window 上也是一样，任何 16 位“宽字符”都是可以接受的文件名，即使字符串不是有效的 UTF-16。操作系统处理的其他字符串也同样如此，比如命令行参数和环境变量。

Rust 字符串始终是有效的 Unicode。实践中的文件名也**几乎**是 Unicode。但 Rust 必须处理那些罕见的可能不是 Unicode 的情形。这就是 Rust 中有 `std::ffi::OsStr` 和 `OsString` 的原因。

`OsStr` 是一种字符串类型，但它是 UTF-8 的超集。它的任务是在当前系统上表示所有文件名、命令行参数和环境变量，**无论它们是不是有效的 Unicode**。在 Unix 上，`OsStr` 可以保存任何字节序列。在 Windows 上，`OsStr` 以 UTF-8 扩展的形式存储，可以编码任何 16 位值的序列，包括不匹配的代理对。

因此我们就有了两个字符串类型：`str` 对应实际的 Unicode 字符串，`OsStr` 对应操作系统可能吐出来的任何东西。此外，`std::path::Path` 也是用于处理文件名的，但它纯粹是出于方便性的考虑。`Path` 与 `OsStr` 其实是一样的，只是增加了很多与文件名相关的便捷方法。接下来的几节会介绍这些方法。`Path` 既可用于绝对路径也可用于相对路径。而对于路径中的个别组件，要使用 `OsStr`。

最后，每种字符串类型都有一个对应的**拥有**类型。比如，`String` 拥有分配在堆上的 `str`，`std::ffi::OsString` 拥有分配在堆上的 `OsStr`，而 `std::path::PathBuf` 拥有分配在堆上的 `Path`，如表 18-1 所示。

表18-1：str、OsStr 与Path

	str	OsStr	Path
非固定大小类型，始终传引用	是	是	是
可以包含任意 Unicode 文本	是	是	是
通常看起来像 UTF-8	是	是	是
可以包含非 Unicode 数据	否	是	是
文本处理方法	是	否	否
文件名相关的方法	否	否	是
拥有型、可增强的堆空间对应类型	`String`	`OsString`	`PathBuf`
转换为拥有类型	`.to_string()`	`.to_os_string()`	`.to_path_buf()`

这 3 个类型全都实现了常用特型 `AsRef<Path>`，因此可以轻松声明一个接收“任何文件名类型”作为参数的泛型函数。这里用到了 13.7 节介绍过的相关技术：

```
use std::path::Path;
use std::io;

fn swizzle_file<P>(path_arg: P) -> io::Result<()>
    where P: AsRef<Path>
{
    let path = path_arg.as_ref();
    ...
}
```

所有以 `path` 为参数的标准函数和方法都使用了这个技术，因此可以给它们直接传字符串字面量。

18.2.2 Path和PathBuf的方法

除了其他未列出的方法外，`Path` 提供了以下方法。

- **`Path::new(str)`** 将 `&str` 和 `&OsStr` 转换为 `&Path`。这样不会复制字符串，新的 `&Path` 只是指向原始 `&str` 和 `&OsStr` 的相同字节。

  ```
  use std::path::Path;
  let home_dir = Path::new("/home/fwolfe");
  ```

  （类似地，`OsStr::new(str)` 将 `&str` 转换为 `&OsStr`。）

- **`path.parent()`** 返回路径的父目录（如果有的话），返回类型是 `Option<&Path>`。

  这样不会复制路径，`path` 的父目录始终是 `path` 的一个子字符串。

  ```
  assert_eq!(Path::new("/home/fwolfe/program.txt").parent(),
             Some(Path::new("/home/fwolfe")));
  ```

- **`path.file_name()`** 返回 `path` 最后的组件（如果有的话），返回类型是 `Option<&OsStr>`。

  通常情况下，`path` 包含目录、斜杠和文件名，因此这个方法返回文件名。

  ```
  assert_eq!(Path::new("/home/fwolfe/program.txt").file_name(),
             Some(OsStr::new("program.txt")));
  ```

- **path.is_absolute()** 和 **path.is_relative()** 用于检测文件是绝对路径还是相对路径，比如 Unix 路径 /usr/bin/advent 或 Windows 路径 C:\Program Files 是绝对路径，而 src/main.rs 是相对路径。
- **path1.join(path2)** 连接两个路径，返回新的 PathBuf。

  ```
  let path1 = Path::new("/usr/share/dict");
  assert_eq!(path1.join("words"),
             Path::new("/usr/share/dict/words"));
  ```

  如果 path2 是绝对路径，则只会返回 path2 的副本，因此这个方法可以用于将任何路径转换为绝对路径：

  ```
  let abs_path = std::env::current_dir()?.join(any_path);
  ```

- **path.components()** 返回给定路径各组件的迭代器，从左到右。迭代项类型是 std::path::Component，这是一个枚举，表示文件名中可能出现的所有可能的组件：

  ```
  pub enum Component<'a> {
      Prefix(PrefixComponent<'a>),  // 仅限Windows：盘符或共享盘
      RootDir,                      // 根目录，/或\
      CurDir,                       // 特殊目录.
      ParentDir,                    // 特殊目录..
      Normal(&'a OsStr)             // 纯文件或目录名
  }
  ```

  例如，Windows 路径 \\venice\Music\A Love Supreme\04-Psalm.mp3 由表示 \\venice 的 Prefix、表示 \Music 的 RootDir 和表示 A Love Supreme 及 04-Psalm.mp3 的两个 Normal 组件组成。

  详细信息请参考在线文档。

这些方法操作的是内存中的字符串。Path 也有一些查询文件系统的方法：.exists()、.is_file()、.is_dir()、.read_dir()、.canonicalize()，等等。更多信息请参考在线文档。

将 Path 转换为字符串有 3 个方法。每个方法都允许 Path 中存在无效的 UTF-8。

- **path.to_str()** 将 Path 转换为字符串，返回类型为 Option<&str>。如果 Path 不是有效的 UTF-8，则返回 None。

  ```
  if let Some(file_str) = path.to_str() {
      println!("{}", file_str);
  } // ……否则跳过这个名字古怪的文件
  ```

- **path.to_string_lossy()** 基本上是一样的，但它总是设法在任何情况下都返回某种字符串。如果 path 不是有效的 UTF-8，则创建一个副本，将每个无效的字节序列替换为 Unicode 替换字符 U+FFFD（�）。

  这个方法的返回类型是 std::borrow::Cow<str>，一个或借用或拥有的字符串。要从这个值取得一个 String，使用它的 .to_owned() 方法。（关于 Cow 的更多信息，参见 13.11 节。）

- **path.display()** 用于打印路径：

  ```
  println!("Download found. You put it in: {}", dir_path.display());
  ```

这个方法返回的值不是字符串，但实现了 `std::fmt::Display`，因此可以在 `format!()`、`println!()` 及类似方法中使用。如果路径不是有效的 UTF-8，则输出中可能包含 � 字符。

18.2.3 文件系统访问函数

表 18-2 展示了 `std::fs` 中的一些函数及其在 Unix 和 Windows 中对应的命令或方法。所有这些函数都返回 `io::Result` 值，除非特别说明，都是 `Result<()>`。

表18-2：文件系统访问函数一览

	Rust函数	Unix	Windows
创建和删除	`create_dir(path)`	`mkdir()`	`CreateDirectory()`
	`create_dir_all(path)`	类似 `mkdir -p`	类似 `mkdir`
	`remove_dir(path)`	`rmdir()`	`RemoveDirectory()`
	`remove_dir_all(path)`	类似 `rm -r`	类似 `rmdir /s`
	`remove_file(path)`	`unlink()`	`DeleteFile()`
复制、移动与链接	`copy(src_path, dest_path) -> Result<u64>`	类似 `cp -p`	`CopyFileEx()`
	`rename(src_path, dest_path)`	`rename()`	`MoveFileEx()`
	`hard_link(src_path, dest_path)`	`link()`	`CreateHardLink()`
检查	`canonicalize(path) -> Result<PathBuf>`	`realpath()`	`GetFinalPathNameByHandle()`
	`metadata(path) -> Result<Metadata>`	`stat()`	`GetFileInformationByHandle()`
	`symlink_metadata(path) -> Result<Metadata>`	`lstat()`	`GetFileInformationByHandle()`
	`read_dir(path) -> Result<ReadDir>`	`opendir()`	`FindFirstFile()`
	`read_link(path) -> Result<PathBuf>`	`readlink()`	`FSCTL_GET_REPARSE_POINT`
权限	`set_permissions(path, perm)`	`chmod()`	`SetFileAttributes()`

（`copy()` 返回的数值表示所复制文件的大小，单位是字节。关于创建符号链接，参见 18.2.5 节。）

可见，Rust 在努力提供跨平台的函数，让它们在 Windows、macOS、Linux 及其他 Unix 系统中都是可预测的。

全面介绍文件系统的特性超出了本书范围，如果你想进一步了解这些函数，可以在网上查到更多相关资料。下一节也会给出几个例子。

所有这些函数都是通过调用操作系统实现的。例如，`std::fs::canonicalize(path)` 不仅仅会通过字符串处理去掉给定 `path` 中的 . 和 ..，它还要基于当前目录解析相对路径，跟踪符号链接。如果 `path` 不存在则会报错。

`std::fs::metadata(path)` 和 `std::fs::symlink_metadata(path)` 产生的 `Metadata` 类型包含有关文件类型、大小、权限和时间戳的信息。同样，详情请参考在线文档。

为方便考虑，`Path` 类型也内置了这样几个方法。比如 `path.metadata()` 其实就相当于 `std::fs::metadata(path)`。

18.2.4　读取目录

要列出目录下的内容，可以使用 `std::fs::read_dir` 或 `Path` 的 `.read_dir()` 方法：

```
for entry_result in path.read_dir()? {
    let entry = entry_result?;
    println!("{}", entry.file_name().to_string_lossy());
}
```

注意，代码中有两行用到了 ? 操作符。第一行要检测打开目录可能遇到的错误。第二行要检测读取下一个条目可能遇到的错误。

这里 `entry` 的类型是 `std::fs::DirEntry`，是一个有几个方法的结构体。

- entry.file_name() 是文件或目录的名称，类型为 `OsString`。
- entry.path() 类似，但包含原始路径，产生一个新 `PathBuf`。如果要读取的目录是 "/home/jimb"，而 entry.file_name() 返回 ".emacs"，那么 entry.path() 会返回 `PathBuf::from`（"/home/jimb/.emacs"）。
- entry.file_type() 返回一个 `io::Result<FileType>`。`FileType` 有 `.is_file()`、`.is_dir()` 和 `.is_symlink()` 方法。
- entry.metadata() 获取当前条目的其他元数据。

在读取目录时，**不会**列出特殊目录 . 和 ..。

下面是一个比较贴近实际的例子。以下代码会在磁盘上将一个目录树递归复制到另一个目录下：

```
use std::fs;
use std::io;
use std::path::Path;

/// 将已有目录src复制到目标路径dst
fn copy_dir_to(src: &Path, dst: &Path) -> io::Result<()> {
    if !dst.is_dir() {
        fs::create_dir(dst)?;
    }

    for entry_result in src.read_dir()? {
        let entry = entry_result?;
        let file_type = entry.file_type()?;
        copy_to(&entry.path(), &file_type, &dst.join(entry.file_name()))?;
    }

    Ok(())
}
```

下面这个函数 `copy_to`，可以复制个别目录的条目：

```
/// 将src中的所有内容复制到目标路径dst
fn copy_to(src: &Path, src_type: &fs::FileType, dst: &Path) -> io::Result<()> {
    if src_type.is_file() {
        fs::copy(src, dst)?;
    } else if src_type.is_dir() {
```

```
        copy_dir_to(src, dst)?;
    } else {
        return Err(io::Error::new(io::ErrorKind::Other,
                                  format!("don't know how to copy: {}",
                                          src.display())));
    }
    Ok(())
}
```

18.2.5　平台特定的特性

前面的 `copy_to` 函数可以复制文件和目录，而我们还想在 Unix 上支持符号链接。

没有跨平台的方法可以同时支持在 Unix 和 Windows 上创建符号链接，但标准库提供了特定于 Unix 的 `symlink` 函数。

```
use std::os::unix::fs::symlink;
```

使用这个函数，可以简化实现。为此只需要给 `copy_to` 函数的 `if` 表达式添加一个分支：

```
...
} else if src_type.is_symlink() {
    let target = src.read_link()?;
    symlink(target, dst)?;
...
```

只要面向 Unix 系统（如 Linux 和 macOS）编译这个程序，就没有问题。

`std::os` 模块包含类似 `symlink` 的各种平台特定的特性。`std::os` 在标准库中实际上类似下面这样（有些像诗，但格律不对）：

```
// ! OS特定功能
#[cfg(unix)]                    pub mod unix;
#[cfg(windows)]                 pub mod windows;
#[cfg(target_os = "ios")]       pub mod ios;
#[cfg(target_os = "linux")]     pub mod linux;
#[cfg(target_os = "macos")]     pub mod macos;
...
```

`\#[cfg]` 属性表示条件编译，其中每个模块都只在特定平台上才会存在。这也是修改后的程序使用 `std::os::unix` 只能面向 Unix 成功编译的原因，因为在其他平台上 `std::os::unix` 是不存在的。

如果想让代码在所有平台上都编译，而且要支持 Unix 平台上的符号链接，则必须也在程序中使用 `\#[cfg]`。此时，只需要在 Unix 上导入 `symlink`，而在其他系统中定义自己的 `symlink` 替代函数：

```
#[cfg(unix)]
use std::os::unix::fs::symlink;

/// 在不支持的平台上定义替代的symlink实现
#[cfg(not(unix))]
fn symlink<P: AsRef<Path>, Q: AsRef<Path>>(src: P, _dst: Q)
```

```
    -> std::io::Result<()>
{
    Err(io::Error::new(io::ErrorKind::Other,
                       format!("can't copy symbolic link: {}",
                               src.as_ref().display())))
}
```

在本书写作时，位于 Rust 官方网站上的在线文档是通过（在 Linux 上）对标准库运行 `rustdoc` 生成的。这意味着其中不包含针对 macOS、Windows 和其他平台特性的内容。查看这些平台特定内容的最好方式是在你自己的平台上运行 `rustup doc`，然后查看生成的 HTML 文档。当然，也可以直接查看源代码。

实际上 `symlink` 是一种特殊情况。大多数特定于 Unix 的特性并非单独的函数，而是给标准库类型添加新方法的扩展特型（11.2.2 节介绍过扩展特型）。有一个 `prelude` 模块，可以用于一次性启用所有这些扩展：

```
use std::os::unix::prelude::*;
```

例如，在 Unix 上，这样会给 `std::fs::Permissions` 添加一个 `.mode()` 方法，可以访问表示 Unix 中权限的一个底层 `u32` 值。类似地，这样也会扩展 `std::fs::Metadata`，增加对底层 `struct stat` 值的访问器字段，比如 `.uid()` 会返回文件所有者的用户 ID。

总的来说，`std::os` 的内容相当基础。更多平台特定的功能都是通过第三方包来支持的，比如可以访问 Windows 注册表的 `winreg`。

18.3 网络编程

关于网络编程的内容超出了本书的范围。不过，假如你已经了解了一些网络编程的内容，那么本节可以帮你尽快上手 Rust 网络编程。

要编写低级的网络代码，可以使用 `std::net` 模块，这个模块提供了对 TCP 和 UPD 网络通信的跨平台支持。要支持 SSL/TLS，可以使用 `native_tls` 包。

这些模块为基于网络创建直观、阻塞式输入和输出提供了方便。可以只用几行代码就实现一个简单的服务器，用 `std::net` 为每个连接创建一个线程。例如，下面就是一个示例的“回声”（echo）服务器：

```
use std::net::TcpListener;
use std::io;
use std::thread::spawn;

// 永远接受连接，每个连接创建一个线程
fn echo_main(addr: &str) -> io::Result<()> {
    let listener = TcpListener::bind(addr)?;
    println!("listening on {}", addr);
    loop {
        // 等待客户端连接
        let (mut stream, addr) = listener.accept()?;
        println!("connection received from {}", addr);
```

```
            // 创建一个线程服务于客户端
            let mut write_stream = stream.try_clone()?;
            spawn(move || {
                // 将从stream中接收的东西再发送回去
                io::copy(&mut stream, &mut write_stream)
                    .expect("error in client thread: ");
                println!("connection closed");
            });
        }
    }

    fn main() {
        echo_main("127.0.0.1:17007").expect("error: ");
    }
```

这个“回声”服务器只是简单地重发你发给它的东西。上面的代码跟使用 Java 或 Python 写的没有多少不同。（下一章会介绍 `std::thread::spawn()`。）

不过，对于高性能服务器，需要异步输入和输出。`mio` 包对此提供了必需的支持。MIO 工作在非常低的级别，提供简单的事件循环和异步读取、写入、连接和接收连接的方法，基本上重现了整个网络 API。每个异步操作完成后，MIO 都会向你写的事件处理程序中发送一个事件。

还有一个实验性的 `tokio` 包，用基于期许（future）的 API 封装了 `mio` 事件循环，期许类似于 JavaScript 中的期约（promise）。

高级协议是通过第三方包支持的。例如，`reqwest` 包为 HTTP 客户端提供了漂亮的 API。下面是一个完整的命令行程序，可以通过以 http 或 https 开头的 URL 获取任何文档，然后在终端打印出来。这些代码使用了 `reqwest = "0.5.1"`。

```
    extern crate reqwest;

    use std::error::Error;
    use std::io::{self, Write};

    fn http_get_main(url: &str) -> Result<(), Box<Error>> {
        // 发送HTTP请求，获得响应
        let mut response = reqwest::get(url)?;
        if !response.status().is_success() {
            Err(format!("{}", response.status()))?;
        }

        // 读取响应体，并将其写入stdout
        let stdout = io::stdout();
        io::copy(&mut response, &mut stdout.lock())?;

        Ok(())
    }

    fn main() {
        let args: Vec<String> = std::env::args().collect();
        if args.len() != 2 {
            writeln!(io::stderr(), "usage: http-get URL").unwrap();
```

```
            return;
        }

        if let Err(err) = http_get_main(&args[1]) {
            writeln!(io::stderr(), "error: {}", err).unwrap();
        }
    }
```

为 HTTP 服务器而写的 `iron` 框架则提供了 `BeforeMiddleware` 和 `AfterMiddleware` 特型等高级协议。这两个特型支持实现可挺拔式应用。而 `websocket` 包实现了 WebSocket 协议。还有很多这样的第三方包，本章就不再列举了。Rust 还是一门年轻的语言，开源社区生机勃勃。支持网络编程的第三方包日新月异、层出不穷。

第19章

并发

> 长远来看，不建议用面向机器的语言编写允许无限使用存储位置及其地址的大型并发程序，因为我们没有办法保证这种程序的可靠性（即使有复杂硬件机制的帮助）。
>
> ——Per Brinch Hansen (1977 年)

> 通信模式就是并行模式。
>
> ——Whit Morriss

如果你对并发的态度在你的职业生涯期间发生了变化，那你并不是唯一一个有这种思想转变的人。这是一种常见现象。

首先，并发代码好写、好玩。线程、锁、队列等工具随取随用。没错，陷阱也很多，但好在我们知道所有这些陷阱，只要小心一点就不会出错。

有些时候，你不得不调试别人写的多线程代码，最终被迫得出结论：**某些人**真的不应该使用这些工具。

而有些时候，你又不得不调试自己的多线程代码。

经验让你养成了对所有多线程代码合理质疑的习惯，甚至会感到彻底灰心。特别是突然看到一篇深入剖析为什么看起来一点问题没有的多线程惯用代码却不能工作的文章，更进一步强化了你的感受。（这与“内存模型”有关。）不过，最终你又发现了一种可以在项目里使用而不会经常出错的并发代码的写法。你可以在这种写法里塞进很多东西，同时（如果你**真的**明白了）也学会了对过多的复杂性说“不”。

当然，惯用写法有很多。系统程序员常用的方法如下。

- 一个**后台线程**（background thread）只负责一件事，而且周期性“醒来”去做这件事。

- 通用**线程池**（worker pool）通过**任务队列**与客户端通信。
- **管道**（pipeline）将数据从一个线程导入另一个线程，每个线程只做一小部分工作。
- **数据并行**（data parallelism）假设（不管正确与否）整个计算机主要用于一项大型计算，这个大型计算进而又拆分成 *n* 个小任务，在 *n* 个线程上执行，希望所有 *n* 个机器的核心同时工作。
- **同步对象海**（sea of synchronized object）中多个线程拥有同一数据权限，使用基于互斥量等低级原语的临时**锁**方案避免争用。（Java 内置支持这种模型，此模型在 20 世纪 90 年代到 21 世纪初一度非常流行。）
- **原子整数操作**（atomic integer operation）允许多核心通过以一个机器字大小的字段传递信息而实现通信。（除非要交换的数据就是整数值，否则这种方法比其他手段更难以保证正确。实践中，这通常意味着传递指针。）

随着时间推移，你可能会用到其中一些方法并将它们安全地组合起来使用。此时你已经算是艺术大师了。只要没有其他人"染指"系统，一切就堪称完美。使用线程的程序尽是无法言说的"潜规则"。

Rust 为使用并发提供了更好的手段，但不是强迫所有程序采用一种风格（这对于系统程序员来说相当于没有解决方案），而是安全地支持多种风格。无法言说的规则在代码中被写了下来，由编译器负责监督。

我们一直在说通过 Rust 可以编写安全、快速的并发程序。因此本章将专门讲述如何开发并发程序，主要介绍了使用 Rust 线程的 3 种方式。

- 并行分叉 – 合并
- 通道
- 共享可修改状态

在此期间，我们会用到迄今为止学习到的关于 Rust 语言的一切。Rust 对引用、可修改能力、生命期的重视在单线程程序里已经体现出了价值，但这些规则只有在并发编程时才真正体现得淋漓尽致。以这些为基础，可以扩展工具箱，可以快速而正确地糅合各种风格的多线程代码，没有质疑，没有灰心，没有恐惧。

19.1　并行分叉–合并

线程的最简单用例是同时执行几个完全无关的任务。

假设要对大量语料文档进行自然语言处理。可以写一个循环：

```
fn process_files(filenames: Vec<String>) -> io::Result<()> {
    for document in filenames {
        let text = load(&document)?; // 读取源文件
        let results = process(text); // 计算统计值
        save(&document, results)?;   // 写入输出文件
    }
    Ok(())
}
```

这个程序的执行过程如图 19-1 所示。

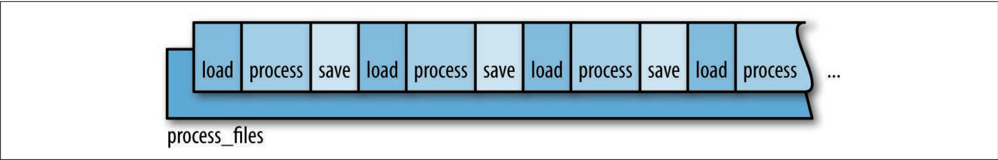


图 19-1：`process_files()` 的单线程执行过程

因为每个文档都是单独处理的，所以要想加快处理速度很容易。只要把语料分块，然后分别在独立的线程上处理每一块即可，如图 19-2 所示。

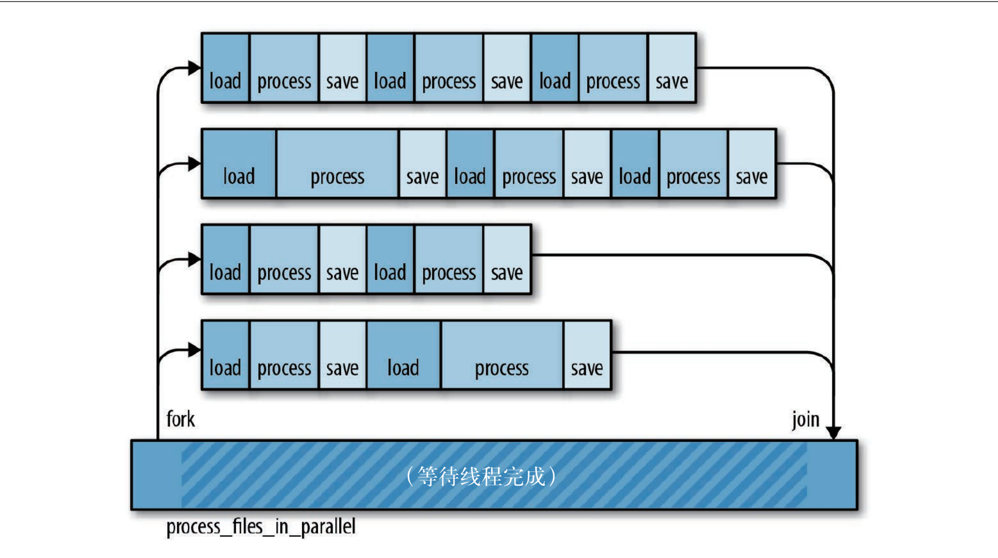


图 19-2：使用分叉 – 合并手段多线程处理文件

这个模式就叫作**并行分叉 – 合并**。**分叉**（fork）就是启动一个新线程，而**合并**（join）就是等待线程完成。第 2 章已经运用过这种技术通过多线程来加速绘制曼德布洛特集合。

并行分叉 – 合并有如下优点。

- 非常简单。分叉 – 合并很容易实现，在 Rust 中也很容易正确处理。
- 避免瓶颈。分叉 – 合并过程中不涉及给共享资源加锁。只有在任务结束后，一个线程才需要等待另一个线程。在执行任务期间，每个线程都可以自由运行。这样可以保证降低任务切换的开销。
- 性能计算直观。在最好的情况下，启动 4 个线程可以只用四分之一时间完成任务。图 19-2 展示了不能对提速抱有理想预期的一个原因，即可能无法为每个线程平均分派任务。另一个要注意的原因是，有时候分叉 – 合并程序在线程合并后必须花一些时间来**组合**所有线程计算的结果。换句话说，完全隔离任务可能造成额外的工作量。不过，除了这两点，任何 CPU 密集型程序在隔离工作单元下都会有显著的性能提升。

- 容易推断程序是否正确。只要线程之间是真正隔离的，分叉 – 合并程序就是**确定性的**，就像绘制曼德布洛特集合的程序一样。无论线程计算速度如何，程序最终都会得到相同结果。这是一种无资源争用的并发模型。

分叉 – 合并的主要缺点是要求工作单元隔离。本章后面会分析一些没办法分得这么清楚的问题。

下面还是以自然语言处理为例，展示几种适用于 `process_files` 函数的分叉 – 合并模式。

19.1.1 产生及合并

要产生一个新线程，可以使用 `std::thread::spawn` 函数。

```
spawn(|| {
    println!("hello from a child thread");
})
```

它接收一个参数、一个 `FnOnce` 闭包或函数。Rust 会启动一个新线程来运行该闭包或函数的代码。这个新线程是一个真实的操作系统线程，有自己的栈，与 C++、C# 和 Java 中的线程一样。

下面是一个更实际的例子，其使用 `spawn` 实现了前面 `process_files` 函数的并行版：

```
use std::thread::spawn;

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // 将工作分成几块
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // 分叉：每个块产生一个线程来处理
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist))
        );
    }

    // 合并：等待所有线程完成
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

下面逐行分析一下这个函数。

```
fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
```

这个并行版本的函数与原始的 `process_files` 有相同的类型签名，因此可以方便替换。

```
// 将工作分成几块
const NTHREADS: usize = 8;
let worklists = split_vec_into_chunks(filenames, NTHREADS);
```

这里使用了一个辅助函数 `split_vec_into_chunks`（没有展示）来拆分工作。结果保存在了 `worklists` 中，这是一个向量的向量，其中包含将原始向量 `filenames` 平均分成的 8 份大小相等的切片。

```
// 分叉：每个块产生一个线程来处理
let mut thread_handles = vec![];
for worklist in worklists {
    thread_handles.push(
        spawn(move || process_files(worklist))
    );
}
```

为每个 `worklist` 产生一个线程。`spawn()` 返回一个名为 `JoinHandle` 的值，后面会用到。这里先把所有 `JoinHandle` 保存到一个向量中。

注意把文件名列表转换为工作线程的过程：

- 在父线程中，通过 `for` 循环定义并转移 `worklist`；
- 创建 `move` 闭包时，`worklist` 被转移到闭包中；
- 然后 `spawn` 将闭包（以及 `worklist` 向量）转移到新的子线程中。

这些转移开销很小。与第 4 章讨论的 `Vec<String>` 的转移一样，`String` 不会被克隆。事实上，整个过程不涉及内存分配和释放。唯一转移的数据是 `Vec` 本身，只有 3 个机器字。

我们创建的每个线程都需要代码和数据来启动。Rust 闭包可以方便地用来包含我们想要的数据和代码。

继续来看代码：

```
// 合并：等待所有线程完成
for handle in thread_handles {
    handle.join().unwrap()?;
}
```

这里使用前面收集的 `JoinHandle` 的 `.join()` 方法来等待全部 8 个线程完成。合并线程对保证正确性是必需的，因为 Rust 程序会在 `main` 返回后立即退出，即使其他线程仍在运行。析构器不会被调用；其他线程会直接被杀死。如果这不是你想要的结果，那就必须在 `main` 返回之前合并这些线程。

如果这个循环结束，就意味着全部 8 个子线程都已经成功完成了。因此函数就返回 `Ok(())` 表示终止：

```
    Ok(())
}
```

19.1.2　跨线程错误处理

因为错误处理，所以前面用来合并子线程的代码比看起来要麻烦。再看一看那行代码：

```
handle.join().unwrap()?;
```

这个 `.join()` 方法为我们做了两件漂亮事。

首先，`handle.join()` 返回一个 `std::thread::Result`，**如果子线程诧异**则是一个错误。相比之下，这就让 Rust 中的线程代码比 C++ 中可靠多了。在 C++ 中，越界访问数组是未定义行为，没有任何保证系统其他部分不受该行为影响的措施。而在 Rust 中，诧异是安全且局限于每个线程的。线程之间的边界构成诧异的防火墙，即诧异不会自动从一个线程传播到依赖它的其他线程。相反，一个线程的诧异在其他线程中会体现为包含错误的 `Result`。程序整体上很容易恢复。

不过在我们的程序中，并没有任何多余的诧异处理代码。我们只是在 `Result` 上立即调用了 `.unwrap()`，断言它是一个 `Ok` 结果，而不是 `Err` 结果。假如某个子线程**确定**诧异了，那么这个断言会失败，因而父线程也会诧异。这里相当于显式将诧异从子线程传播到父线程。

其次，`handle.join()` 把子线程返回的值传给了父线程。我们传给 `spawn` 的闭包的返回类型是 `io::Result<()>`，也就是 `process_files` 返回值的类型。这个返回值不会被丢弃。在子线程完成时，其返回值会被保存，而 `JoinHandle::join()` 会将该值传送到父线程。

在这个程序中，`handle.join()` 返回的完整类型是 `std::thread::Result<std::io::Result<()>>`，其中的 `thread::Result` 部分是 `spawn/joinAPI` 相关类型，`io::Result` 是我们应用的相关类型。

对我们的代码来说，这里在展开 `thread::Result` 之后，对 `io::Result` 使用了 `?` 操作符，显式将 I/O 错误从子线程传播到父线程。

这些看起来好像还挺复杂的。但毕竟只是一行代码，因此可以跟其他语言比较一下。Java 和 C# 的默认行为是将子线程的异常抛到终端，然后就不管了。在 C++ 中，默认行为是中断进程。而在 Rust 中，错误是一种 `Resut` 值（数据），而不是异常（控制流）。可以像其他任何值一样跨线程传送它们。不论何时，只要编写使用低级线程 API 的代码，就必须仔细编写错误处理代码。**既然必须要编写错误处理代码**，那么使用 `Result` 没错。

19.1.3 跨线程共享不可修改数据

假设我们要做的分析需要一个大型英语单词和短语的数据库：

```
// 之前
fn process_files(filenames: Vec<String>)

// 之后
fn process_files(filenames: Vec<String>, glossary: &GigabyteMap)
```

这里的 `glossary` 将会很大，因此我们传它的引用。如何更新 `process_files_in_parallel` 把这个词汇表传给工作线程呢？

这样明显的修改是不行的：

```
fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: &GigabyteMap)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
```

```
        thread_handles.push(
            spawn(move || process_files(worklist, glossary)) // 错误
        );
    }
    ...
}
```

我们简单地给函数添加了一个 glossary 参数，直接把它传给 process_files。Rust 会抱怨：

```
error[E0477]: the type `[closure@...]` does not fulfill the required lifetime
  --> concurrency_spawn_lifetimes.rs:35:13
   |
35 |             spawn(move || process_files(worklist, glossary))  // 错误
   |             ^^^^^
   |
   = note: type must satisfy the static lifetime
```

Rust 抱怨的是传给 spawn 的闭包的生命期。

spawn 会启动一个独立的线程。Rust 无法知道一个子线程会运行多长时间，因此它假设一种最坏的情况，即子线程可能会在父线程已经完成且父线程中所有的值都消失之后继续运行。显然，如果子线程会持续如此长的时间，那么它运行的闭包也需要持续这么长的时间。但这个闭包的生命期是界定的，它依赖 glossary 引用，而引用不会永远存在。

注意，Rust 拒绝编译这个代码是对的！按照编写这个函数的方式，如果一个线程触发了 I/O 错误，就可能导致 process_files_in_parallel 在其他线程完成前退出。那么其他子线程就有可能在主线程被释放之后还继续使用 glossary。这就造成了争用，假如判主线程赢的话，那奖品就是未定义行为。Rust 不会允许这种情况发生。

看起来 spawn 在支持跨线程引用方面是很开放的。确实，14.1.2 节介绍过类似情形。当时的解决方案是使用 move 闭包把数据的所有权转移到新线程。但这里不行，因为有多个线程需要使用同一份数据。为每个线程克隆一份词汇表是一种安全的方案，但因为词汇表太大，我们不能这么做。好在标准库提供了另一种方式：原子引用计数。

4.4 节介绍过 Arc，现在该把它派上用场了：

```
use std::sync::Arc;

fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: Arc<GigabyteMap>)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        // 这里调用.clone()只是克隆Arc并触发引用计数。并不会克隆GigabyteMap
        let glossary_for_child = glossary.clone();
        thread_handles.push(
            spawn(move || process_files(worklist, &glossary_for_child))
        );
    }
    ...
}
```

为此我们修改了 `glossary` 的类型：为并行运行分析，调用者必须传入一个 `Arc<GigabyteMap>`。这是一个指向通过 `Arc::new(giga_map)` 移动到堆上的 `GigabyteMap` 的智能指针。

调用 `glossary.clone()` 后，会创建 `Arc` 智能指针而不是整个 `GigabyteMap` 的一个副本。这相当于增加一次引用计数。

这样修改之后，程序就可以编译通过并运行了，因为它不再依赖引用的生命期。只要有**任何**线程拥有 `Arc<GigabyteMap>`，映射就不会释放，即使父线程早就退出了。因为 `Arc` 中的数据是不可修改的，所以也不会出现任何数据争用。

19.1.4 Rayon

标准库的 `spawn` 函数是一个重要的原语，但并不是专门为并行分叉 – 合并设计的。已经有更好的分叉 – 合并 API 构建在其基础之上。例如，第 2 章使用的 Crossbeam 库在 8 个线程间分配任务。Crossbeam 的**受限线程**（scoped thread）可以相当自然地支持并行分叉 – 合并。

Niko Matsakis 写的 Rayon 库是另一个例子。这个库提供了两种运行并发任务的方式：

```
extern crate rayon;
use rayon::prelude::*;

//“并行做两件事”
let (v1, v2) = rayon::join(fn1, fn2);

//“并行做N件事”
giant_vector.par_iter().for_each(|value| {
    do_thing_with_value(value);
});
```

`rayon::join(fn1, fn2)` 就是调用两个函数并返回两个结果。而 `.par_iter()` 方法会创建一个 `ParallelIterator`，这个值有 `map`、`filter` 和其他方法，非常类似于 Rust 的 `Iterator`。在上面的两种情况下，Rayon 会使用自己的工作线程池在必要时拆分工作。只要告诉 Rayon 什么任务**可以**并行去做，Rayon 就会尽可能以最佳的方式管理线程和任务分配。

图 19-3 展示了两种理解 `giant_vector.par_iter().for_each(...)` 调用的方式。(a) 表面上看，Rayon 会为向量中的每个元素都启动一个线程。(b) 在后台，Rayon 会让每个工作线程对应一个 CPU 核心，这样效率更高。这个工作线程池由程序的所有线程共享。在同时有数千个任务时，Rayon 会自动拆分工作。

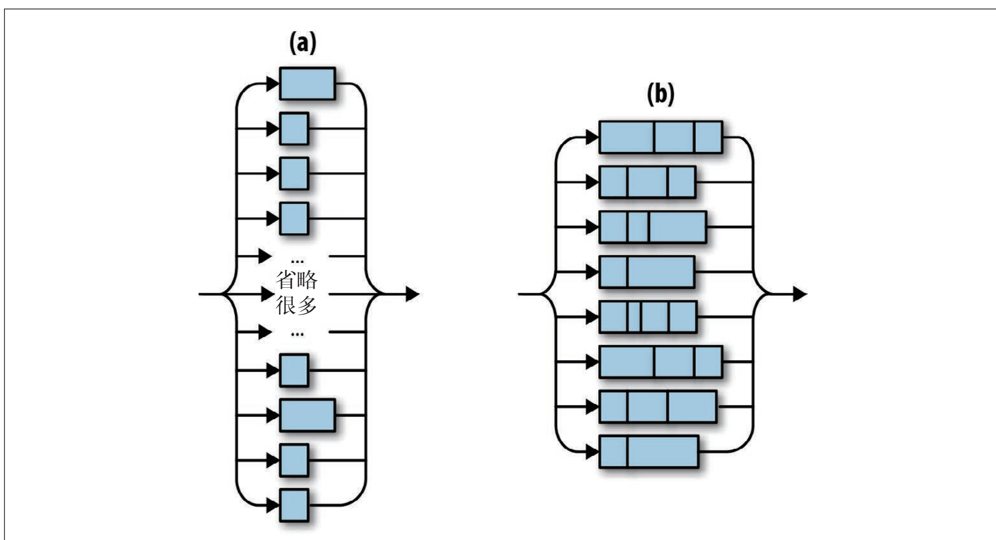


图 19-3：Rayon 的理论和实践

这是使用 Rayon 重写的 process_files_in_parallel 的版本：

```
extern crate rayon;

use rayon::prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &GigabyteMap)
    -> io::Result<()>
{
    filenames.par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| {
            if r1.is_err() { r1 } else { r2 }
        })
        .unwrap_or(Ok(()))
}
```

相比于 std::thread::spawn 的版本，以上代码更少，也更好理解。下面逐行分析一下。

- 首先，使用 filenames.par_iter() 创建并行迭代器。
- 使用 .map() 对每个文件名（filename）调用 process_file。这样会得到 io::Result<()> 值的一个 ParallelIterator。
- 然后用 .reduce_with() 组合结果。在这里，我们保留第一个错误（如果有的话），然后丢弃其他错误。如果想累积所有错误或者打印它们，可以在这里操作。

  也可以给 .reduce_with() 方法传入一个成功后返回有用值的 .map() 闭包。或者给 .reduce_with() 传入一个知道如何组合两个成功结果的闭包。

- .reduce_with() 返回一个 Option，只有 filename 为空时才是 None。最后使用 Option 的 .unwrap_or() 方法让结果 Ok(())。

在后台，Rayon 使用一种名为**工作窃取**（work-stealing）的技术动态地在线程间平衡负载。通常，这样会比 19.1.1 节展示的手工提前分派工作效率更高，更能保持 CPU 满载。

不仅如此，Rayon 还支持在线程间共享引用。任何后台发生的并行处理都可以保证在 `reduce_with` 返回时完成。这也是可以把 `glossary` 传给 `process_file`，而不用考虑该闭包会被多个线程调用的原因。

（顺便说一下，这里使用 `map` 和 `reduce` 方法并不是巧合。由 Google 和 Apache Hadoop 带火的 MapReduce 编程模型与分叉 – 合并有很多共通之处。可以将其看作一种查询分布式数据的分叉 – 合并方法。）

19.1.5 重温曼德布洛特集合

第 2 章使用分叉 – 合并方式并发渲染了曼德布洛特集合。结果渲染速度快了 4 倍，非常令人震撼。不过还没有达到它应有的震撼程度。假如在一个 8 核的机器上启动 8 个工作线程呢？

问题在于我们做不到工作负载的平均分配。计算图像中的一个像素相当于运行一个循环（参见 2.6.1 节）。而图像浅灰色区域由于循环会快速退出而比黑色部分渲染得更快。因此，虽然把图像分成了相同大小的水平长条，但实际上每一条的工作负载是不均等的，如图 19-4 所示。

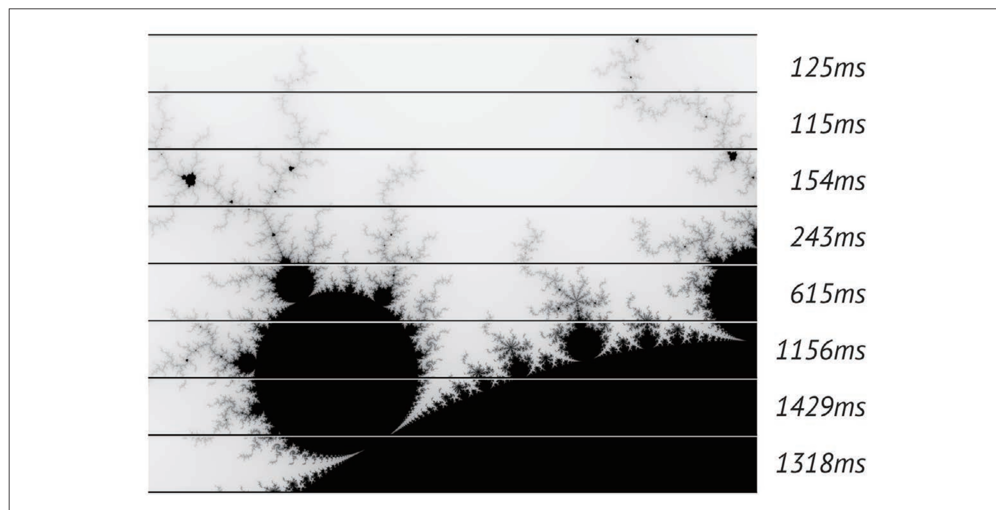


图 19-4：曼德布洛特程序中不均等的工作分布

使用 Rayon 很容易解决这个问题。可以将输出中的每一行像素作为一个并行任务，于是就可以创建数百个任务。Rayon 可以将这些任务分派给自己的线程。由于工作窃取技术，每个任务工作量的大小都不重要了。Rayon 会在计算期间动态平衡。

下面是实现代码。第一行和最后一行是 2.6.6 节展示过的 `main` 函数的一部分，中间部分是修改后的渲染代码。

```
let mut pixels = vec![0; bounds.0 * bounds.1];

// 将pixels切分为水平长条的作用域
{
    let bands: Vec<(usize, &mut [u8])> = pixels
        .chunks_mut(bounds.0)
        .enumerate()
        .collect();

    bands.into_par_iter()
        .weight_max()
        .for_each(|(i, band)| {
            let top = i;
            let band_bounds = (bounds.0, 1);
            let band_upper_left = pixel_to_point(bounds, (0, top),
                                                 upper_left, lower_right);
            let band_lower_right = pixel_to_point(bounds, (bounds.0, top + 1),
                                                upper_left, lower_right);
        render(band, band_bounds, band_upper_left, band_lower_right);
    });
}

write_bitmap(&args[1], &pixels, bounds).expect("error writing PNG file");
```

首先，创建要传给 Rayon 的任务集合 `bands`。每个任务就是一个 `(usize, &mut [u8])` 元组，包含计算所需的行号和对应的 `pixels` 中的切片。这里使用 `chunks_mut` 方法把图片缓冲区拆成行，而 `enumerate` 给每一行添加行号，`collect` 再把所有编号的切片对收集到向量中。（因为 Rayon 只能基于数组或向量创建并行迭代器，所以这里需要创建一个向量。）

接下来，把 `bands` 转换为一个并行迭代器，调用 `.weight_max()` 方法告诉 Rayon 这些任务非常占用 CPU 资源，然后再使用 `.for_each()` 方法告诉 Rayon 我们想完成什么工作。

因为要使用 Rayon，所以 main.rs 中必须加入以下代码：

```
extern crate rayon;
use rayon::prelude::*;
```

`Cargo.toml` 也要加入：

```
[dependencies]
rayon = "0.4"
```

经过如此一番修改，程序在 8 核机器上大约会使用 7.75 个核心。因而速度较之前手工拆分任务时会提升 75%，而且代码量更少。这反映出了让包来干活儿（分派任务）而不是我们自己干的好处。

19.2　通道

通道（channel）是把值从一个线程发送到另一个线程的单向管道。换句话说，它是一个线程安全的队列。

图 19-5 展示了通道的用法。通道与 Unix 中的管道有点类似，都是一端发送数据，另一端接

收数据，而且这两端通常由不同的线程所有。但 Unix 管道发送的是字节，而通道发送的是 Rust 值。`sender.send(item)` 把一个值放进通道，`receiver.recv()` 则移除一个值。所有权也从发送线程转移到了接收线程。如果通道是空的，`receiver.recv()` 则会一直阻塞到有值发送。

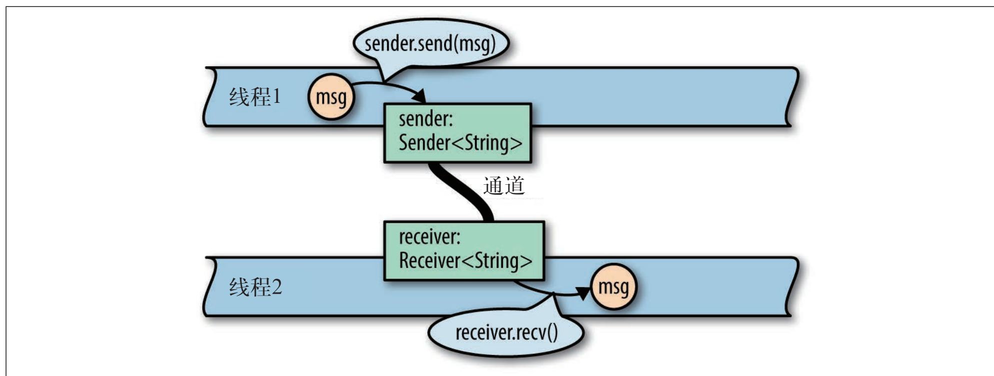


图 19-5：字符串通道。字符串 `msg` 的所有权从线程 1 转移到了线程 2

使用通道，线程可以通过传值实现通信。这是线程间协作的一种很简单的方式，无须使用锁或者共享内存。

这不是一个新技术。Erlang 拥有隔离进程和消息传递已经 30 年了。Unix 管道也已经存在将近 50 年了。一般来说，我们会将管道理解为一种提供灵活性和复合能力、但不并发的特性，而实际上，管道可以实现上面的所有功能。图 19-6 展示了一个 Unix 管道的示例，其中涉及的所有 3 个程序是完全可以同时运行的。

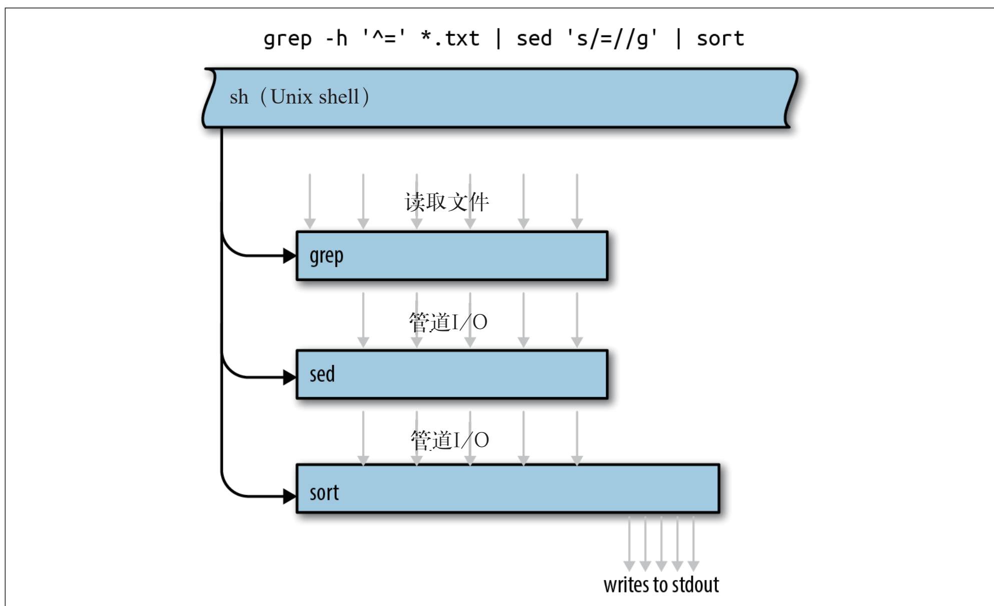


图 19-6：Unix 管道执行流程

Rust 通道比 Unit 管道快。发送值是转移而不是复制，而转移的值即使数据结构包含几兆数据也是很快的。

19.2.1　发送值

接下来几节，我们会使用通道构建一个并发程序，这个程序会创建一个**倒排索引**（inverted index），倒排索引是搜索引擎的关键要素之一。每个搜索引擎都工作在特定文档集合之上。倒排索引是一种数据库，可以查询哪个关键词在哪里出现过。

本书中会展示与线程和通道相关的部分代码，完整代码可参见本书在 GitHub 网站上的页面。这个程序不长，总共大约 1000 行代码。

示例程序是按照管道的思路搭建的，如图 19-7 所示。管道只是使用通道的多种方式之一，后面还会讨论其他几种方式。不过，管道是在已有单线程程序中加入并发的直观方式。

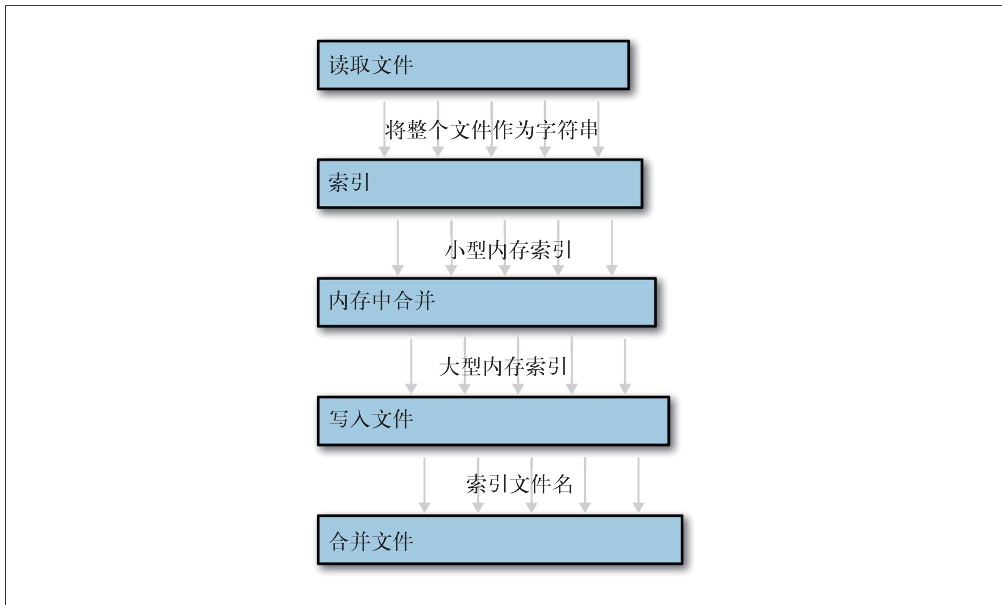


图19-7：索引构建器管道。箭头表示通过管道把值从一个线程发送到另一个线程。没有展示磁盘I/O部分

这个程序总共使用了 5 个线程来分别完成不同的任务。每个线程在程序的生命期内都会不断产生输出。比如，第一个线程负责从磁盘把源文档一个接一个地读取到内存中。（专门用一个线程来做这件事是因为我们想把代码写得尽量简单，使用的 `File::open` 和 `read_to_string` 也都是阻塞 API。我们不想在磁盘忙碌时 CPU 却无所事事。）这一阶段的输出是每个文档对应一个长 `String`，因此这个线程与下一个线程是通过 `String` 通道连接的。

程序将从启动读取文件的线程开始。假设 `documents` 保存的是 `Vec<PathBuf>`，即文档名向量。启动读取文件线程的代码如下所示：

```
use std::fs::File;
use std::io::prelude::*; // 为使用Read::read_to_string
use std::thread::spawn;
use std::sync::mpsc::channel;

let (sender, receiver) = channel();

let handle = spawn(move || {
    for filename in documents {
        let mut f = File::open(filename)?;
        let mut text = String::new();
        f.read_to_string(&mut text)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});
```

通道是 `std::sync::mpsc` 模块的一部分，稍后再解释这个名字的含义。下面来看看代码的工作过程，先从创建通道开始：

```
let (sender, receiver) = channel();
```

这里的 `channel` 函数返回一对值：发送者（sender）和接收者（receiver）。底层实现是一个队列数据结构，标准库隐藏了实现细节。

通道是有类型的。我们想使用这个通道发送每个文件的文本，因此 `sender` 和 `receiver` 的类型分别是 `Sender<String>` 和 `Receiver<String>`。当然，也可以显式地声明要创建字符串通道，比如这样写：`channel::<String>()`。Rust 的类型推断可以做这件事。

```
let handle = spawn(move || {
```

跟以前一样，启动线程要使用 `std::thread::spawn`。`sender`（而非 `receiver`）的所有权会通过这个 `move` 闭包转移给新线程。

接下来的几行代码仅用于从磁盘读取文件：

```
for filename in documents {
    let mut f = File::open(filename)?;
    let mut text = String::new();
    f.read_to_string(&mut text)?;
```

读取文件成功后，要把其文本内容发送给通道：

```
    if sender.send(text).is_err() {
        break;
    }
}
```

`sender.send(text)` 把值 `text` 转移给通道。最终，这个值会被转移给接收它的对象。无论 `text` 包含的是 10 行文本还是 10 兆字节，这个操作都只涉及复制 3 个机器字（`String` 的大小），而对应的 `receiver.recv()` 调用也会只复制 3 个机器字。

send 和 recv 方法都返回 Result，但这两种方法都会在通道另一端被清除的情况下失败。send 调用会在 Receiver 被清除时失败，否则值就会在通道里待一辈子：没有 Receiver，任何线程都没办法接收它。类似地，recv 调用会在通道里没有待接收的值且 Sender 被清除时失败，否则 recv 就要永远等下去：没有 Sender，任何线程都没办法再发送下一个值。清除通道端点是在使用完它之后"挂起"并关闭连接的正常方式。

在我们的代码中，sender.send(text) 只会在接收者的线程提前退出时失败。这通常考虑的是使用通道的代码。无论这种情况是有意造成的还是错误导致的，读取线程静默地关闭自己是完全没问题的。

当这种情况发生时，或者在线程读取完所有文档后，程序返回 Ok(())：

```
        Ok(())
    });
```

注意，这个闭包返回了一个 Result。如果线程遇到了 I/O 错误，则会立即退出，而错误会存储在线程的 JoinHandle 中。

当然，与其他编程语言一样，Rust 承认错误处理有很多其他的可能方式。在错误发生时，可以通过 println! 将其打印输出，然后再切换到下一个文件，也可以使用发送数据的通道发送错误，把通道作为 Result 通道，或者专门为错误再创建一个通道。而示例程序中采用的方式既轻量又可靠：? 操作符可以避免大量样板代码，甚至像 Java 的 try/catch 那样的错误处理代码，而且错误也不会静默传递。

为方便起见，我们把所有这些代码包装在一个函数中，这个函数同时返回 receiver（还没有用到）和新线程的 JoinHandle：

```
fn start_file_reader_thread(documents: Vec<PathBuf>)
    -> (Receiver<String>, JoinHandle<io::Result<()>>)
{
    let (sender, receiver) = channel();

    let handle = spawn(move || {
        ...
    });

    (receiver, handle)
}
```

注意，这个函数启动新线程后立即返回了。我们会为管道中每一阶段都写一个类似这样的函数。

19.2.2 接收值

本书前面实现了在一个线程里循环发送值。接下来可以再创建第二个线程来循环调用 receiver.recv()：

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

不过 `Receiver` 本身是可迭代的，因此更简单的方式是这样写：

```
for text in receiver {
    do_something_with(text);
}
```

这两个循环是等价的。无论怎么写，当控制流到达循环顶部时，只要通道恰好为空，接收线程就会阻塞，直到其他线程发送值。循环会在通道为空且 `Sender` 被清除的情况下正常退出。在我们的程序中，读取线程退出后循环会自然正常退出。读取线程会运行一个闭包，该闭包拥有变量 `sender`，闭包退出时，`sender` 会被清除。

下面来写管道第二阶段的代码：

```
fn start_file_indexing_thread(texts: Receiver<String>)
    -> (Receiver<InMemoryIndex>, JoinHandle<()>)
{
    let (sender, receiver) = channel();

    let handle = spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id, text);
            if sender.send(index).is_err() {
                break;
            }
        }
    });
    (receiver, handle)
}
```

这个函数创建了一个线程，该线程从一个通道（`texts`）接收 `String` 值并向另一个通道（`sender`/`receiver`）发送 `InMemoryIndex` 值。这个线程的工作是获取第一阶段加载的每个文件，把每个文档转换为一个小型单文件、驻内存的倒排索引。

这个线程的主循环很好理解。索引文档的全部工作都由 `from_single_document` 函数完成。这里就不展示它的源代码了，不过把输入字符串按照单词切分，然后再生成单词与位置列表的映射并不难。

这一阶段不执行 I/O，因此不是必须要处理 `io::Error`。为此，这个函数没有返回 `io::Result<()>`，而是返回了 `()`。

19.2.3 运行管道

剩下的 3 个阶段在设计上也是类似的。每个阶段都要消费上一阶段创建的 `Receiver`。对管道的其余部分，我们的目标是把所有小索引合并为一个大索引文件，然后保存在磁盘上。达成这个目标至少需要 3 个阶段。这里就不再展示具体代码实现了，详情可以查看在线文档。下面只看看这 3 个阶段对应函数的类型签名。

首先，在内存中合并索引直至足够大（第三阶段）：

```
fn start_in_memory_merge_thread(file_indexes: Receiver<InMemoryIndex>)
    -> (Receiver<InMemoryIndex>, JoinHandle<()>)
```

然后，把大索引写入磁盘（第四阶段）：

```
fn start_index_writer_thread(big_indexes: Receiver<InMemoryIndex>,
                             output_dir: &Path)
    -> (Receiver<PathBuf>, JoinHandle<io::Result<()>>)
```

最后，如果有多个大文件，则使用基于文件的合并算法将它们合并起来（第五阶段）：

```
fn merge_index_files(files: Receiver<PathBuf>, output_dir: &Path)
    -> io::Result<()>
```

最后这个阶段不会返回 `Receiver`，因为它是管道的终点。这个阶段会在磁盘上生成一个输出文件。它也不返回 `JoinHandle`，因为这个阶段也不需要再创建线程了。调用线程决定工作何时完成。

接下来看一看启动线程和检查错误的代码：

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
    -> io::Result<()>
{

    // 启动管道的全部5个阶段
    let (texts,   h1) = start_file_reader_thread(documents);
    let (pints,   h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files,   h4) = start_index_writer_thread(gallons, &output_dir);
    let result = merge_index_files(files, &output_dir);

    // 等待线程完成，保存碰到的任何错误
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();

    // 返回遇到的第一个错误（如果有的话）。
    //（此时，h2和h3不会失败，因为这些线程是纯内存数据处理。）
    r1?;
    r4?;
    result
}
```

跟以前一样，使用 `.join().unwrap()` 显式地把诧异从子线程传播到主线程。不同之处是这里没有马上使用 `?`，而是把 `io::Result` 值放在一边，直到 4 个线程全部合并完成。

相比于单线程设备，这个管道要快 40%。对于一个下午就能完成的工作来说，这个结果还不错。但相比于曼德布洛特程序几乎 75% 的提速就有点相形见绌了。显然，我们不是没有充分利用系统的 I/O 容量，就是没有充分利用所有的 CPU 内核。这是怎么回事呢？

管道与制造工厂的流水线类似，其整体性能受限于最慢阶段的生成能力。全新的、没有调优过的流水线可能跟装配单件产品一样慢。但流水线的优势在定向调优后会发挥出来。在我们的例子中，数据显示第二个阶段是瓶颈。索引线程使用的是 `.to_lowercase()` 和 `.is_alphanumeric()`，因此要在搜索 Unicode 表上花费很多时间。索引之后的下游阶段把大多数时间花在了 `Receiver::recv` 休眠、等待输入上。

这意味着还可以更快。只要解决瓶颈问题，并行度就可以提升。现在，我们已经知道如何使用通道，而我们的程序也由隔离的代码片段构成，因此很容易找到解决第一个瓶颈的方法。可以手工优化第二阶段的代码，就像优化其他代码一样，可以把工作拆分成两个或更多个阶段，或者同时运行多个文件索引线程。

19.2.4 通道特性与性能

`std::sync::mpsc` 中的 `mpsc` 指的是“multi-producer, single-consumer”（**多生产者，单消费者**），是对 Rust 通道所提供通信特性的简洁概括。

示例程序中的通道只从一个发送者取值，发送给一个接收者。这是相当常见的情况。但 Rust 通道也支持多个发送者，此时由一个线程处理来自多个客户端线程的请求，如图 19-8 所示。

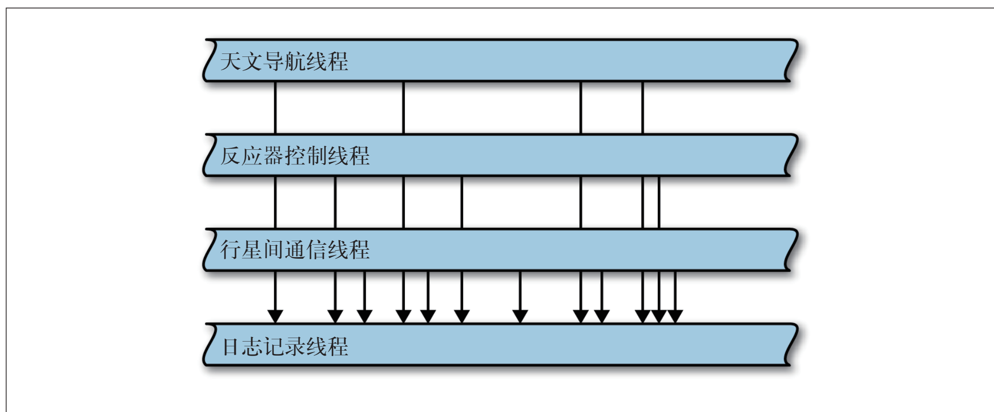


图 19-8：一个通道接收多个发送者的请求

`Sender<T>` 实现了 `Clone` 特型。要获得一个有多个发送者的通道，只需创建一个常规通道，然后再克隆发送者，需要多少就克隆多少。可以把每个 `Sender` 值转移到不同的线程。

`Receiver<T>` 无法克隆，因此如果需要多个线程从同一个通道接收值，就需要使用 `Mutex`。本章后面会就此给出例子。

Rust 通道是经过认真优化的。在刚创建通道时，Rust 使用的是“一次性”队列实现。如果只是用这个通道发送一个对象，那可以保证开销最小。如果再发送第二个值，Rust 则会切换到一个不同的队列实现。这个实现会从长远考虑，准备让通道传输很多值，同时又保持分配开销最小化。如果你选择克隆 `Sender`，Rust 则必须回退到另外一个实现，该实现可以保证多个线程同时发送值时的安全。不过即使是这 3 个实现中最慢的实现也是没有锁的队列，因此发送和接收值最多只是几个原子操作，涉及一次堆内存分配，外加转移自身。只有在队列为空且接收线程因此需要休眠时才需要系统调用。当然，此时经过通道的流量无论如何也不是最大的。

除了上述这些优化工作，对应用程序来说，还有一个在通道性能方面很可能会犯的错误：

发送值的速度超过接收和处理值的速度。这会导致通道内部的值越积越多。例如，对我们的程序而言，我们发现文件读取器线程（阶段一）加载文件的速度可能远快于文件索引线程（阶段二）索引它们的速度。结果导致从磁盘读取了几百兆字节的原始数据，一次性全装进了队列里。

这种错误占用内存且损害局部性。更糟糕的是，如果发送线程一直运行，则会耗尽 CPU 和其他系统资源以发送更多值，而接收端此时正急需这些资源却得不到。

这里 Rust 同样借鉴了 Unix 管道。Unix 使用了一种优雅的技巧来提供某种**反压力**（backpressure），从而强迫快速发送端放慢速度。Unix 系统的每个管道都有固定大小，如果一个进程尝试向随时可能满的管道写入数据，系统就会直接阻塞该进程，直至管道中有了空间。Rust 中的等价机制叫**同步通道**（synchronous channel）。

```
use std::sync::mpsc::sync_channel;

let (sender, receiver) = sync_channel(1000);
```

同步通道就像常规通道一样，只是在创建时需要指定它可以保存多少值。对于同步通道而言，`sender.send(value)` 是一个潜在的阻塞操作。毕竟，阻塞也不总是坏事。在示例程序中，当把 `start_file_reader_thread` 中的 `channel` 改为可以保存 32 个值的 `sync_channel` 后，根据我们使用基准数据测试，在保持吞吐量不变的情况下，可以降低三分之二内存占用。

19.2.5　线程安全：Send与Sync

目前为止，我们一直假定所有值都可以在进程间自由转移和共享。大多数情况下确实如此，但 Rust 代码的彻底安全取决于两个内置特型：`std::marker::Send` 和 `std::marker::Sync`。

- 实现 `Send` 的类型可以安全地把值传到另一个线程，即它们可以在线程间转移。
- 实现 `Sync` 的类型可以安全地把不可修改引用传到另一个线程，即它们可以在线程间共享。

这里所谓的**安全**，就是我们一直反复强调的意思：没有数据争用和其他未定义行为。

例如，在 19.1.1 节 `process_files_in_parallel` 的例子中，我们使用闭包从父线程向每个子线程传了一个 `Vec<String>`。当时并没有指出这一点，但这意味着向量及其字符串的分配是在父线程中，释放则在子线程中。由于 `Vec<String>` 实现了 `Send`，因此 API 可以保证这是没问题的，即 `Vec` 内部使用的分配程序和 `String` 是线程安全的。

（如果使用快速但非线程安全的分配程序来写自己的 `Vec` 和 `String` 类型，那就必须使用非 `Send` 类型实现它们，例如不安全的指针。Rust 可以推断出你的非线程安全的向量 `NonThreadSafeVec` 和非线程安全的字符串 `NonThreadSafeString` 不是 `Send`，并限制它们只能在单线程中使用。但这种情况很少见。）

如图 19-9 所示，大多数类型既是 `Send` 也是 `Sync`。在程序中，甚至都不必使用 `\#[derive]` 为结构体和枚举来实现这两个特型。Rust 会自动帮你实现。结构体或枚举的字段如果是 `Send`，那它们也是 `Send`；如果是 `Sync`，那它们也是 `Sync`。

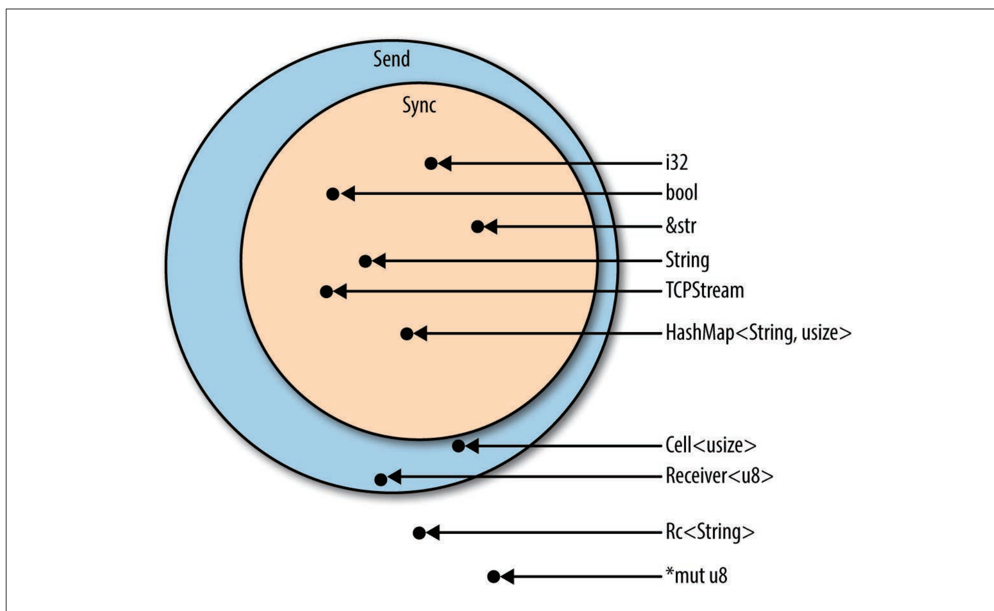


图 19-9：Send 和 Sync 类型

少数没有实现 Send 和 Sync 的类型主要用于在非线程安全的条件下提供可修改能力。例如引用计数智能指针类型 std::rc::Rc<T>。

如果可以在线程间共享 Rc<String> 会导致什么问题？如果两个线程恰好同时克隆这个 Rc，如图 19-10 所示，就会出现数据争用，因为两个线程都会增加共享引用计数。结果引用计数可能会变得不准确，从而导致将来的“释放后还使用”或双重释放，这都是未定义行为。

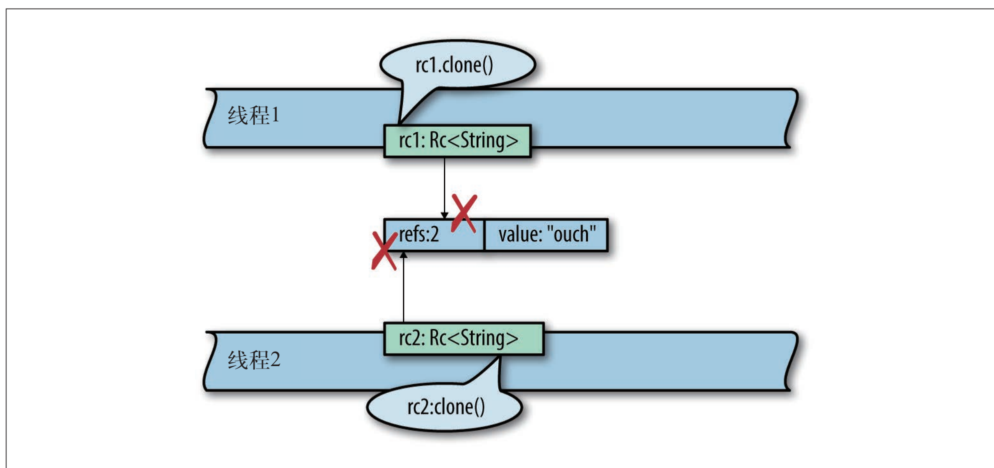


图 19-10：为什么 Rc<String> 既不是 Sync 也不是 Send

当然，Rust 会阻止这样做。以下代码故意造成了数据争用：

```
use std::thread::spawn;
use std::rc::Rc;

fn main() {
    let rc1 = Rc::new("hello threads".to_string());
    let rc2 = rc1.clone();
    spawn(move || {  // 错误
        rc2.clone();
    });
    rc1.clone();
}
```

Rust 拒绝编译这段代码，并给出了详细的错误消息：

```
error[E0277]: the trait bound `Rc<String>: std::marker::Send` is not satisfied
              in `[closure@...]`
  --> concurrency_send_rc.rs:10:5
   |
10 |     spawn(move || {  // 错误
   |     ^^^^^ within `[closure@...]`, the trait `std::marker::Send` is not
   |           implemented for `Rc<String>`
   |
   = note: `Rc<String>` cannot be sent between threads safely
   = note: required because it appears within the type `[closure@...]`
   = note: required by `std::thread::spawn`
```

现在，我们已经知道 Send 和 Sync 可以帮助 Rust 实现线程安全。在跨线程传输数据的函数如 spawn 中，它们都会作为绑定出现在类型签名中。在通过 spawn 创建线程时，传入的闭包必须是 Send。这意味着它所包含的所有值都必须是 Send。类似地，如果要把值从通道中发送给另一个线程，那这个值也必须是 Send。

19.2.6 将所有迭代器都接到通道上

我们示例中的倒排索引构建器是作为一个管道来构建的。代码本身一目了然，但需要手工创建通道并启动线程。相比之下，第 15 章构建的迭代器管道好像是把很多工作打包到了寥寥几行代码中。可以为线程管道构建类似的东西吗？

事实上，如果可以统一迭代器管道和线程管道当然很好。这样索引构建器就可以写成迭代器管道了。可能一开始是这样的：

```
documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)   // 过滤错误结果
    .off_thread()              // 为上面的工作创建一个线程
    .map(make_single_file_index)
    .off_thread()              // 为阶段二创建另一个线程
    ...
```

特型支持给标准库类型添加方法，因此实际上我们可以做到这样。为此首先要写一个特型，声明想要的方法：

```
use std::sync::mpsc;

pub trait OffThreadExt: Iterator {
    /// 把这个迭代器转换为一个分离线程迭代器：
    /// 在一个独立的工作线程上调用next()，
    /// 因此这个迭代器和循环体是并发运行的
    fn off_thread(self) -> mpsc::IntoIter<Self::Item>;
}
```

然后为迭代器类型实现这个特型。`mpsc::Receiver` 已经是可迭代类型，因此会方便一些。

```
use std::thread::spawn;

impl<T> OffThreadExt for T
    where T: Iterator + Send + 'static,
          T::Item: Send + 'static
{
    fn off_thread(self) -> mpsc::IntoIter<Self::Item> {
        // 创建一个通道并将工作线程中的项传出来
        let (sender, receiver) = mpsc::sync_channel(1024);

        // 把这个迭代器转移到一个新线程中并在那里运行它
        spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });
        // 返回一个从通道提取值的迭代器
        receiver.into_iter()
    }
}
```

代码中的 `where` 子句是通过非常类似于 11.5 节描述的一个流程来确定的。一开始只有以下这些内容：

```
impl<T: Iterator> OffThreadExt for T
```

这也就是说，我们希望这个实现适用于所有迭代器。Rust 并没有实现这个。因为我们使用 `spawn` 把一个类型 `T` 的迭代器转移到了新线程，所以必须指定 `T: Iterator + Send + 'static`。因为要把项从通道中传出来，所以必须指定 `T::Item: Send + 'static`。有了这些绑定，Rust 就没有意见了。

下面简单总结一下 Rust 的性格：我们可以随意给语言的所有迭代器添加并发能力，但前提是必须理解并明确指定保证它们安全使用的限制。

19.2.7　超越管道

本节以管道为例，因为管道是使用通道的一个既自然又明显的方式。所有人都能理解管道和通道，因为它们具体、实际且具有确定性。不过，通道不仅在管道中有用，它们也是在相同进程中为其他线程提供异步服务的快速而简单的方式。

假设你想用一个线程来记录日志，如图 19-8 所示的那样。其他线程可以借助通道向日志线程发送消息。因为可以克隆通道的 `Sender`，所以很多客户端线程可以有向同一个日志线程投递消息的发送者。

在自己的线程上运行类似记录日志这样的服务有很多好处。日志线程可以在必要时更替日志文件。为此，它无须与其他线程有过多沟通。那些线程不会被锁定。消息可以无害地在通道中累积片刻，直到日志线程恢复工作。

通道也适用于一个线程向另一个线程发送请求并期待得到某种响应的情形。第一个线程的请求可以是一个结构体或元组，包含一个 `Sender`，一个标明发件人地址的信封，以便第二个线程用它回寄响应。这并不意味着交互必须同步。第一个线程决定是阻塞并等待响应，还是使用 `.try_recv()` 方法轮询结果。

目前为止我们展示了针对适合高度并行计算任务的分叉 – 合并，还有适合松散连接组件的通道。这两种并发机制可以满足很多应用的需求。但本章内容不止这些，下面继续。

19.3　共享可修改状态

在第 8 章发布 `fern_sim` 包几个月之后，你的蕨类植物模拟程序已经真正走红了。现在，你打算再写一个多人实时策略游戏，让 8 个玩家在模拟的侏罗纪世界中比赛种植生长周期接近真实世界的蕨类植物。这个游戏的服务器是一个大规模并行应用，拥有来自众多线程的大量请求。在 8 个玩家都上线之后，这些线程之间是如何协调来开始游戏的？

这里要解决的问题是很多线程需要访问等待加入游戏的玩家的共享列表。这个数据必须在所有线程中都可以修改且共享。如果 Rust 没有共享的可修改状态，那该怎么办呢？

可以为此创建一个新线程，其全部工作就是管理这个列表。其他线程可以使用通道与其通信。当然，这要占用一个需要一些操作系统开销的线程。

另一个选项是使用 Rust 为安全地共享可修改数据所提供的工具。没错，确实有这个选项。它们就是所有熟悉线程的系统程序员都知道的低级元语。本节会介绍互斥量、读 / 写锁、条件变量和原子整数。最后会展示如何在 Rust 中实现全局可修改变量。

19.3.1　什么是互斥量

互斥量（或者叫**锁**）用于强制多线程依次访问特定的数据。下一节会介绍 Rust 的互斥量。首先，有必要回忆一下其他语言中的互斥量是什么样的。下面是在 C++ 中简单使用互斥量的一个场景：

```
// C++代码，不是Rust
void FernEngine::JoinWaitingList(PlayerId player) {
    mutex.Acquire();

    waitingList.push_back(player);

    // 如果等待的玩家满足条件则开始游戏
    if (waitingList.length() >= GAME_SIZE) {
        vector<PlayerId> players;
```

```
            waitingList.swap(players);
            StartGame(players);
        }
        mutex.Release();
    }
```

代码中调用 `mutex.Acquire()` 和 `mutex.Release()` 的语句是**临界区**（critical section）的开始和结束。对程序中的每个 `mutex`，每次只有一个线程可以在临界区中运行。如果临界区中有一个线程，则所有调用 `mutex.Acquire()` 的其他线程都会被阻塞，直至第一个线程调用 `mutex.Release()`。

我们说，互斥量**保护**数据。对这里而言，就是 `mutex` 保护 `waitingList`。不过，确保每个线程在访问数据前获得互斥量，之后再释放它则是程序员的责任。

互斥量的作用体现在以下几方面。

- 防止**数据争用**，即避免多个线程并发读写同一块内存。数据争用在 C++ 和 Go 中属于未定义行为。Java 和 C# 等托管语言承诺不会崩溃,但数据争用的结果仍然（说到底）没有意义。
- 即使没有数据争用，即使所有读写在程序中都是顺序执行，如果没有互斥量，不同线程的操作也可能以任意方式相互交错。想象一下在其他线程可以修改你程序数据的情况下写代码的情景。再想象一下怎么调试这个程序。这样的程序就好像中邪了。
- 互斥量支持通过**不变性**（invariant）编程，即受保护数据由你负责初始化但由每个临界区来维护的规则。

当然，所有这些实际上都基于相同的原因：不受控的争用情形会导致编程困难。互斥量为混乱中引入了秩序（尽管没有通道或分叉 – 合并那么有秩序）。

不过，在大多数语言中，互斥量很容易搞乱套。在 C++ 中（在其他大多数语言中也一样），数据和锁是两个独立对象。理想情况下，注释会解释每个线程必须在触碰数据前先获得互斥量：

```
class FernEmpireApp {
    ...

private:
    // 等待加入游戏的玩家列表。通过mutex来保护
    vector<PlayerId> waitingList;

    // 在读写waitingList之前必须先获得锁
    Mutex mutex;
    ...
};
```

即使有了这些贴心的注释，编译器照样无法保证这里的安全访问。如果一段代码忘了先获得互斥量，就会出现未定义行为。实践中，这意味着极难重现和修复的错误。

就算在 Java 中对象和互斥量之间存在某些概念上的关联，但这种关联仍然不够牢固。编译器会尝试保持这种关联性。而在实践中，由锁保护的数据很少恰好是关联对象的字段，其中经常包含来自多个对象的数据。锁机制照样难以捉摸。因此注释仍然是落实这种关联的主要工具。

19.3.2　Mutex<T>

现在来看看在 Rust 中如何实现等待列表。在我们的“蕨类帝国”（Fern Empire）游戏服务器上，每个用户都有一个唯一的 ID：

```
type PlayerId = u32;
```

等待列表就是一个玩家 ID 的集合：

```
const GAME_SIZE: usize = 8;

/// 等待列表不会超过GAME_SIZE个玩家
type WaitingList = Vec<PlayerId>;
```

等待列表作为 `FernEmpireApp` 的一个字段来保存，它是在服务器启动时在 `Arc` 中初始化的一个单例对象。每个线程都有一个指向它的 `Arc`。它包含所有共享配置及程序所需的其他东西，其中大多数是只读的。因为等待列表既是共享的也是可修改的，所以必须由一个 `Mutex` 来提供保护：

```
use std::sync::Mutex;

/// 所有线程都可以共享访问这个大上下文结构体
struct FernEmpireApp {
    ...
    waiting_list: Mutex<WaitingList>,
    ...
}
```

与 C++ 不同，Rust 中受保护的数据保存在 `Mutex` **内部**。创建 `Mutex` 的代码是这样的：

```
let app = Arc::new(FernEmpireApp {
    ...
    waiting_list: Mutex::new(vec![]),
    ...
});
```

创建一个新 `Mutex` 就像创建一个新 `Box` 或 `Arc`，但 `Box` 和 `Arc` 都意味着堆分配，而 `Mutex` 就是单纯的一种锁。如果想把 `Mutex` 分配在堆上，则必须明确地表示出来，就像这里使用 `Arc::new` 创建整个应用，而使用 `Mutex::new` 只是为了保护数据一样。这两个类型经常一块使用，`Arc` 方便跨线程共享数据，而 `Mutex` 方便跨线程共享可修改数据。

接下来实现使用这个互斥量的 `join_waiting_list` 方法：

```
impl FernEmpireApp {
    /// 向等待列表中为下一个游戏添加一名玩家
    /// 如果等待的玩家够了就立即启动新游戏
    fn join_waiting_list(&self, player: PlayerId) {
        // 锁住互斥量并获得数据访问权
        // 这里guard的作用域是临界区
        let mut guard = self.waiting_list.lock().unwrap();

        // 现在执行游戏逻辑
        guard.push(player);
```

```
            if guard.len() == GAME_SIZE {
                let players = guard.split_off(0);
                self.start_game(players);
            }
        }
    }
```

取得数据唯一的方式是调用 .lock() 方法：

```
    let mut guard = self.waiting_list.lock().unwrap();
```

self.waiting_list.lock() 会一直阻塞到可以再次获得互斥量。这个方法调用返回的 MutexGuard<WaitingList> 值是对 &mut WaitingList 的一个简单封装。借助 13.5 节介绍的 Deref 类型转换，可以直接在这个守卫（guard）上调用 WaitingList 方法：

```
    guard.push(player);
```

这个守卫甚至还允许我们直接引用底层数据。Rust 的生命期系统保证这些引用的寿命不会超出守卫自身。如果没有拿到锁，则不可能在 Mutex 中访问数据。

在 guard 被清除后，锁也会被释放。通常这会在阻塞结束时发生，但也可以手工清除：

```
    if guard.len() == GAME_SIZE {
        let players = guard.split_off(0);
        drop(guard);   // 开始游戏时，清除守卫
        self.start_game(players);
    }
```

19.3.3 mut与Mutex

join_waiting_list 方法没有以 self 的 mut 引用作为参数，这好像有点奇怪（乍一看确实是很奇怪）。这个方法的类型签名是：

```
    fn join_waiting_list(&self, player: PlayerId)
```

底层集合 Vec<PlayerId> **确实**需要一个 mut 引用才能调用 push 方法，其类型签名为：

```
    pub fn push(&mut self, item: T)
```

可是这个代码还是可以编译运行。到底怎么回事？

在 Rust 中，mut 意味着**专有 / 排他访问**（exclusive access）。非 mut 才意味着**共享访问**（shared access）。

我们已经习惯了从父线程向子线程、从容器向内容传递 mut 引用。如果你本来就有对 starships 的 mut 引用（或者拥有 starships 的所有权，此时要恭喜你成为 Elon Musk），就可以在 starships[id].engine 上调用 mut 方法。这是默认行为，因为如果在父上下文中没有专有访问权，那 Rust 通常没法保证你在子上下文中有专有访问权。

但 Mutex 有一个办法：锁。事实上，互斥量就是一个锁，它提供对其中数据的专有（mut）访问权，即使很多线程有对 Mutex 本身的**共享**（非 mut）访问权。

Rust 的类型系统会告诉我们 Mutex 做了什么：它动态控制专有访问。而这通常是由 Rust 编译器在编译时静态完成的。

（你可能还记得 std::cell::RefCell 也会这样做，只不过没有考虑支持多线程。Mutex 和 RefCell 都是内部修改能力的体现，9.9 节曾介绍过。）

19.3.4 互斥量的问题

在讨论互斥量之前，本书先介绍了几种实现并发的方式，而这些方式即使在 C++ 中也可以轻松正确地使用。这并非巧合，因为这些方式对解决并发编程最令人头疼的问题是非常安全且有保证的。只依赖并行分叉 – 合并的程序具有确定性，不可能死锁。而使用通道的程序几乎也不会出什么问题。专门使用通道实现管道操作的程序（比如前面示例中的索引构建器）也具有确定性，虽然消息传输的时间可能不同，但并不影响输出。还有很多这样的例子。多线程程序的安全有保证是很令人欣慰的。

Rust 中 Mutex 的设计几乎肯定可以让你比以往任何时候都能更系统、更明智地使用互斥量。但是有必要停下脚步，思考一下 Rust 的安全保证什么时候有用，什么时候帮不上忙。

安全的 Rust 代码不会触发**数据争用**，即多线程并发读取同一块内存导致产生无意义的结果。这非常好，数据争用毫无疑问是个问题，而且在实际的多线程程序中也并不鲜见。

可是，使用互斥量的线程也会伴有其他一些 Rust 无法帮我们解决的问题。

- 有效的 Rust 程序不会出现数据争用，但仍然可能存在**竞态条件**（race condition），即程序行为取决于线程的执行时间，因此每次运行的结果可能都不相同。有些竞态条件是良性的，有些则表现为常见且极难修复的 bug。以非结构化方式使用互斥量会导致竞态条件。确保竞态条件是良性的由你决定。
- 共享的可修改状态也会影响程序设计。通道作为代码中抽象的边界为隔离组件、方便测试提供了基础，而互斥量鼓励“添加一个方法”式的解决问题的思路，这种思路会导致纠缠不清难以剥离的代码。
- 最后，互斥量并不像它们乍看起来那么简单，接下来的两节会进一步讨论。

所有这些问题都是工具本身所固有的。记住，要尽可能使用结构化方式，而在必需时再使用 Mutex。

19.3.5 死锁

线程在尝试获取自己已经持有的锁时可能会造成死锁：

```
let mut guard1 = self.waiting_list.lock().unwrap();
let mut guard2 = self.waiting_list.lock().unwrap();  // 死锁
```

假设第一次调用 self.waiting_list.lock() 成功，取得了锁。第二次调用发现锁已经被持有了，因此就会阻塞，等待释放。于是它就会永远等待下去。这个等待线程就是持有锁的线程。

从另一个角度说，Mutex 中的锁不是递归锁。

这里代码的问题很明显。在真实程序里，两次 lock() 调用可能出现在不同方法中，其中一个会调用另一个。单独来看，每个方法的代码都很好。还有其他情况可能导致死锁，比如多个线程同时获取多个互斥量。Rust 的借用系统不能保护你避免死锁。最好的保护是保持临界区最小化：进入，干活儿，退出。

使用通道也有可能导致死锁。例如，两个线程可能相互阻塞，每个都等待从另一个接收消息。不过，良好的程序设计可以让你高度自信现实中不会发生这种情况。在管道中，比如我们的倒排索引构建器中，数据流是非循环的。就跟在 Unix 管道中一样，在这样的程序里不可能发生死锁。

19.3.6　中毒的互斥量

Mutex::lock() 返回一个 Result，原因与 JoinHandle::join() 一样：如果另一个线程诧异了，则可以优雅地收场（失败）。在写 handle.join().unwrap() 时，我们是在告诉 Rust 把诧异从一个线程传播到另一个线程。常用的 mutex.lock().unwrap() 也一样。

如果线程在持有 Mutex 时诧异了，那么 Rust 会将 Mutex 标记为**已中毒**。后续想要锁住这个受污染的 Mutex 的尝试都会得到一个错误结果。我们的 .unwrap() 调用告诉 Rust 在这种情况下要诧异，把其他线程的诧异传播到当前线程。

如果出现了中毒，那互斥量的结果有多糟糕呢？中毒听起来很吓人，但具体情况不一定是致命的。正如第 7 章所说的，诧异是安全的。诧异的线程保证了程序其余部分处在安全状态。

基于诧异毒化互斥量并不是因为害怕未定义行为，而是担心你有可能正在使用不变性编程。由于程序诧异了，没有完成应该做的事就脱离了临界区，因此可能已经更新了受保护数据的某些字段，但尚未更新其他字段。为此，原先的不变性可能已经遭到破坏。Rust 通过毒化这个互斥量来防止其他线程在不经意间也出现这种局面，从而避免把问题搞得更糟糕。在完全互斥的情况下，还是**可以**锁住中毒的互斥量并访问其中数据的。具体信息可以参考 PoisonError::into_inner() 的在线文档。不过你不会意外做到这一点的。

19.3.7　使用互斥量的多消费者通道

前面提到过，Rust 通道是多生产者、单消费者的。或者更具体地说，一个通道只有一个 Receiver。任何线程池都不能有多个线程使用一个 mpsc 通道共享工作成果。

不过，有个非常简单的方式可以绕过这个限制，只需要使用标准库。可以为 Receiver 添加一个 Mutex，然后再共享。下面是一个实现它的模块：

```
pub mod shared_channel {
    use std::sync::{Arc, Mutex};
    use std::sync::mpsc::{channel, Sender, Receiver};

    /// 对Receiver的线程安全的封装
    #[derive(Clone)]
    pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);
```

```
    impl<T> Iterator for SharedReceiver<T> {
        type Item = T;

        /// 从封装的接收者获取下一项
        fn next(&mut self) -> Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }
    /// 创建一个新通道，其接收者可以跨线程共享。这会返回一个
    /// 发送者和一个接收者，与stdlib的channel()类似，有时候
    /// 可以直接代替它使用
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
        let (sender, receiver) = channel();
        (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
    }
}
```

这里使用了一个 `Arc<Mutex<Receiver<T>>>`，其中泛型被嵌套了很多层。这种情况在 Rust 中比在 C++ 中更常见。好像这样更容易让人迷惑，但通常在这种情况下，只要依次读出它们的名字就可以理解其含义，如图 19-11 所示。

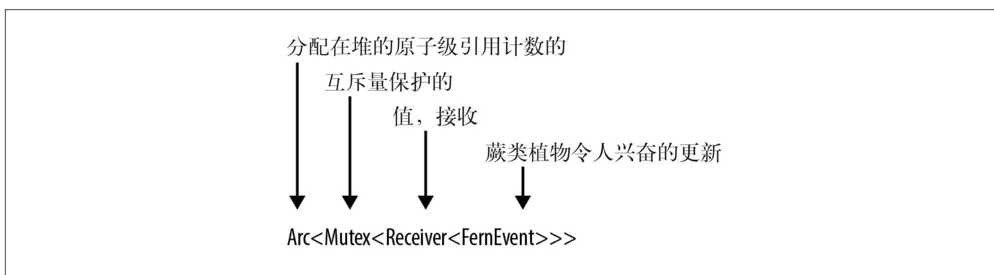


图 19-11：嵌套的泛型

19.3.8 读/写锁（RwLock<T>）

下面介绍 Rust 标准库工具箱提供的另一个线程同步工具：`std::sync`。我们会快速介绍，因为对这些工具的全面讨论超出了本书范围。

服务器程序经常有一些配置信息，这些信息只需加载一次，之后很少会改变。大多数线程只查询这个配置，但由于配置还是**有可能**会改变（比如请求服务器从磁盘重新加载其配置），因此必须通过一个锁来保护。在类似这样的情况下，使用互斥量是可以的，但是个不必要的瓶颈。如果不是正在被修改，线程就不应该排队来查询配置。这时候可以使用**读 / 写锁**或 `RwLock`。

互斥量有一个 `lock` 方法，读 / 写锁则有两个：`read` 和 `write`，其中，`RwLock::write` 方法与 `Mutex::lock` 类似，都是等待获取对受保护数据的专有 `mut` 访问。而 `RwLock::read` 方法提供了非 `mut` 访问，其优点是不太可能需要等待，因为多个线程可以同时安全读取数据。在使用互斥量的情况下，在任意给定时刻，受保护数据只能有一个读取器或写入器（或者两

者都没有)。在使用读 / 写锁的情况下，则可以有一个写入器或多个读取器，这看起来非常像 Rust 的引用。

`FernEmpireApp` 可以有一个结构体保存配置信息，由一个 `RwLock` 来保护：

```
use std::sync::RwLock;

struct FernEmpireApp {
    ...
    config: RwLock<AppConfig>,
    ...
}
```

读取这个配置的方法可以使用 `RwLock::read()`：

```
/// 如果应该使用实验性真菌代码则为true
fn mushrooms_enabled(&self) -> bool {
    let config_guard = self.config.read().unwrap();
    config_guard.mushrooms_enabled
}
```

重新加载这个配置的方法应该使用 `RwLock::write()`：

```
fn reload_config(&self) -> io::Result<()> {
    let new_config = AppConfig::load()?;
    let mut config_guard = self.config.write().unwrap();
    *config_guard = new_config;
    Ok(())
}
```

当然，Rust 非常到位地对 `RwLock` 数据施加了安全限制。这个单写入器或多读取器的概念是 Rust 借用系统的核心。`self.config.read()` 返回一个守卫，可以提供对 `AppConfig` 的非 `mut`（即共享）访问；`self.config.write()` 返回一个不同类型的守卫，可以提供 `mut`（即专有）访问。

19.3.9　条件变量（Condvar）

一个线程经常需要等待某个条件变为 `true`。

- 在服务器关机期间，主线程可能需要等待所有其他线程完全退出。
- 工作线程在没什么事可做时，需要等待要处理的数据。
- 实现分布式共识协议的线程可能需要等待足够多对等线程的响应。

有时候，针对某个需要等待的条件会有方便的阻塞 API，比如对服务器关机的例子有 `JoinHandle::join`。但有时候没有内置的阻塞 API。此时程序可以使用**条件变量**（condition variable）来构建自己的 API。在 Rust 中，`std::sync::Condvar` 类型实现了条件变量。`Condvar` 有方法 `.wait()` 和 `.notify_all()`，其中，`.wait()` 可以阻塞到某些线程调用 `.notify_all()`。

不过问题还会更复杂一些，因为条件变量始终代表由某个 `Mutex` 保护的数据或真或假的条件。为此，`Mutex` 和 `Condvar` 是有关系的。由于篇幅所限，本书对此就不作完整的解释了。不过对于之前使用过条件变量的程序员，我们可以展示两段关键代码。

在期待的条件变为 true 时，调用 Condvar::notify_all（或 notify_one）以唤醒任何等待的线程：

```
self.has_data_condvar.notify_all();
```

要进入休眠并等待某个条件变为 true，使用 Condvar::wait()：

```
while !guard.has_data() {
    guard = self.has_data_condvar.wait(guard).unwrap();
}
```

这个 while 循环是等待条件变量的标准写法。不过，Condvar::wait 的签名非同寻常：它接收 MutexGuard 对象的值，然后消费它，再在成功时返回一个新的 MutexGuard。这给人的感觉是 wait 方法释放了互斥量，然后在返回之前又重新获得了它。传入 MutexGuard 的值相当于表明：“.wait() 方法，我把释放互斥量的特权授予你。”

19.3.10 原子类型

std::sync::atomic 模块包含无锁并发编程要使用的原子类型。这些类型基本上与标准 C++ 原子类型相同。

- AtomicIsize 和 AtomicUsize 是共享的整数类型，对应单线程的 isize 和 usize 类型。
- AtomicBool 是一个共享的 bool 值。
- AtomicPtr<T> 是不安全指针类型 *mut T 的共享值。

关于如何正确使用原子数据超出了本书范围。一言以蔽之，就是多线程可以同时读取原子值但不会导致数据争用。

与常规算术和逻辑操作符不同，原子类型暴露了执行**原子操作**的方法，涉及个别值的加载、存储、交换和算术操作，作为一个单元执行，而且即使其他线程也对同一块内存执行原子操作仍能保证安全。比如，下面的代码会递增名为 atom 的 AtomicIsize：

```
use std::sync::atomic::Ordering;

atom.fetch_add(1, Ordering::SeqCst);
```

这些方法可以编译为特殊的机器语言指令。在 x86-64 架构上，.fetch_add() 调用编译为 lock incq 指令，其中普通的 n += 1 可以编译为纯 incq 指令或相关的其他任何变体。Rust 编译器同样必须放弃对原子操作的某些优化，因为（与正常加载、存储不同）其他线程可以理所当然地立即观察到。

参数 Ordering::SeqCst 是一个**内存排序**（memory ordering）。内存排序类似于数据库中的一个事务隔离层。它们告诉系统相比于性能，你对这些哲学概念（比如导致前面的效果和没有循环的时间）有多么关注。内存排序对程序执行是否正确至关重要，但也非常不好理解和推断。不过令人高兴的是，选择顺序一致性（最严格的内存排序）的性能损失通常很低。这跟把 SQL 数据库放到 SERIALIZABLE 模式下的性能损失不可同日而语。因此如果不敢肯定，那就用 Ordering::SeqCst。Rust 从标准 C++ 原子类型继承了其他几种内存排序，对存在和时间也有不同程度的弱化保证。这里就不讨论它们了。

原子的一个最简单应用就是取消操作。假设有一个线程正在执行某个耗时的计算任务，比如渲染视频，而我们希望能够异步取消这个操作。问题在于如何与希望其停止工作的线程通信。可以通过一个共享的 AtomicBool 来实现：

```
use std::sync::atomic::{AtomicBool, Ordering};

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

以上代码创建了两个 Arc<AtomicBool> 智能指针，都指向分配在堆上的 AtomicBool，其初始值是 false。第一个名为 cancel_flag，会保留在主线程。第二个名为 worker_cancel_flag，会被转移到工作线程。

下面就是工作线程相关的代码：

```
let worker_handle = spawn(move || {
    for pixel in animation.pixels_mut() {
        render(pixel); // 光线跟踪，需要花几微秒时间
        if worker_cancel_flag.load(Ordering::SeqCst) {
            return None;
        }
    }
    Some(animation)
});
```

渲染完每个像素，线程都会调用 .load() 方法检查取消标志的值：

```
worker_cancel_flag.load(Ordering::SeqCst)
```

如果主线程决定取消工作线程的工作，就可以把 true 保存到 AtomicBool，然后等着线程自己退出：

```
// 取消渲染
cancel_flag.store(true, Ordering::SeqCst);

// 丢弃结果，有可能是None
worker_handle.join().unwrap();
```

当然，还有其他实现方式。比如，可以用 Mutex<bool> 或通道来代替这里的 AtomicBool。主要区别在于原子的开销最小。原子操作永远不使用系统调用。加载和存储经常编译为一个 CPU 指令。

原子是一种内部修改能力，与 Mutex 或 RwLock 类似，因此它们的方法也以 self 的共享（非 mut）引用为参数。而这也让它们可以作为简单的全局变量来使用。

19.3.11　全局变量

假设我们正在编写网络代码。我们希望有一个全局变量，一个计数器，每次发送一个数据包都给它递增一次：

```
/// 服务器成功处理的数据包数
static PACKETS_SERVED: usize = 0;
```

这可以编译通过。不过有个问题：PACKETS_SERVED 是不可修改的，因此无法修改它。

Rust 尽其所能阻止全局可修改状态。当然使用 const 声明的常量是不可修改的。静态变量默认同样不可修改，因此无法取得某个值的 mut 引用。static 可以声明为 mut，但再访问它就是不安全的。Rust 的这些规则的主要目的都是保证线程安全。

全局可修改状态同样会带来严重的软件工程后果，比如造成程序不同部分更紧密耦合，更难测试，将来也更难修改。同样，在某些情况下确实没有合理的替代，因此最好能找到一种安全的方式来声明可修改的静态变量。

支持递增 PACKETS_SERVED，同时又能保证线程安全的最简单方式，就是把它改成一个原子整数：

```
use std::sync::atomic::{AtomicUsize, ATOMIC_USIZE_INIT};

static PACKETS_SERVED: AtomicUsize = ATOMIC_USIZE_INIT;
```

这里的常量 ATOMIC_USIZE_INIT 是一个值为 0 的 AtomicUsize。之所以使用这个常量而不是表达式 AtomicUsize::new(0)，是因为静态原子变量的初始值必须是常量。到 Rust 1.17 为止，还不允许方法调用。类似地，ATOMIC_ISIZE_INIT 是一个值为 0 的 AtomicIsize，而 ATOMIC_BOOL_INIT 是一个值为 false 的 AtomicBool。

在这个静态被声明之后，递增包计数器就很简单了：

```
PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

原子全局变量只能是简单的整数或布尔值。不过，要创建其他任何类型的全局变量也都需要解决两个问题，都很容易。

- 变量必须通过某种方式保证线程安全，因为要不然就不能是全局变量。考虑到安全，静态变量必须既是 Sync 又是非 mut。

  幸运的是，我们已经看到过这个问题的解决方案了。Rust 有针对安全共享可变化值的类型：Mutext、RwLock 和原子类型。这些类型即使在被声明为非 mut 的情况下也是可以修改的。这就是它们的目的所在。（参见 19.3.3 节。）

- 如前所述，静态初始化器不能调用函数。这意味着声明静态 Mutex 的显式方式行不通。

```
static HOSTNAME: Mutex<String> =
    Mutex::new(String::new()); // 错误：声明静态值时调用函数
```

可以使用 lazy_static 包来解决这个问题。

17.5.2 节介绍过 lazy_static 包。通过 lazy_static! 宏定义的变量可以使用任何表达式来初始化。这个表达式会在变量第一次被解引用时运行，而值会保存下来供后续操作使用。

可以像下面这样使用 lazy_static 来声明一个全局 Mutex：

```
#[macro_use] extern crate lazy_static;

use std::sync::Mutex;
```

```
lazy_static! {
    static ref HOSTNAME: Mutex<String> = Mutex::new(String::new());
}
```

同样的技术也可以用于 RwLock 和 AtomicPtr 变量。

使用 lazy_static! 会在每次访问静态数据时造成微小的性能损失，因为其实现使用了为一次性初始化而设计的一个低级同步原语 std::sync::Once。在后台，每次访问懒静态数据，程序都要执行一次原子加载指令以检查初始化是否完成。（Once 是有特殊用途的，这里不作详细介绍。通常为了方便起见还是应该使用 lazy_static!。不过在初始化非 Rust 库时，Once 很有用。对此请参见 21.8.5 节的例子。）

19.4　习惯编写Rust并发代码

本章介绍了 3 种在 Rust 中使用线程的技术：并行分叉 – 合并、通道和基于锁共享可修改状态。本章的目的是全面介绍 Rust 为此提供的能力，并以贴近实际应用为出发点。

Rust 坚持线程安全，因此从你决定编写多线程程序那一刻开始，焦点就是构建安全、结构化的通信。保持线程隔离很大程度上可以让 Rust 相信你的代码是安全的。而且隔离也是保证你的代码正确运行和容易维护的基础。同样，Rust 会指导你写出优秀的程序。

更重要的是，Rust 允许你同时使用多种技术，允许实验。你可以快速迭代：面对编译器质疑而不断改进然后上线要比调试数据争用的效率高太多了。

第20章

宏

> 集句（cento，源自拉丁文 patchwork，意为“拼凑的东西”）是一种完全从另一个人的诗句里摘录的句子拼成的诗。
>
> ——Matt Madden

> 这是你引用的名言。
>
> ——Bjarne Stroustrup

Rust 支持宏。**宏**是扩展语言的一种方式，可以实现超越函数的功能。例如，我们已经见过的 `assert_eq!` 宏，可以方便地用于测试：

```
assert_eq!(gcd(6, 10), 2);
```

这可以写成一个泛型函数，不过 `assert_eq!` 宏可以做到函数做不到的几件事。一件事是在断言失败时，`assert_eq!` 宏可以生成包含断言文件名和行号的错误消息。函数没办法取得这些信息。宏可以，因为它的工作方式与函数完全不同。

宏是某种简写形式。在编译期间，在检查类型和生成任何机器码之前，每个宏调用都会被**扩展**（expanded）。换句话说，每个宏调用都会被替换成其他一些 Rust 代码。比如前面的宏调用扩展后就是这样的：

```
match (&gcd(6, 10), &2) {
    (left_val, right_val) => {
        if !(*left_val == *right_val) {
            panic!("assertion failed: `(left == right)`, \
                    (left: `{:?}`, right: `{:?}`)", left_val, right_val);
        }
    }
}
```

其中的 panic! 也是一个宏，因此它之后也会扩展为某些 Rust 代码。这些代码会使用另外两个宏，即 file!() 和 line!()。在包中所有的宏调用全部扩展完毕后，Rust 才会进入下一个编译阶段。

在运行时，断言失败会像下面这样（而且会表示 gcd() 函数中存在 bug，因为 2 是正确的返回值）：

```
thread 'main' panicked at 'assertion failed: `(left == right)`, (left: `17`,
right: `2`)', gcd.rs:7
```

如果你有 C++ 编程经验，可能会对宏有一些坏印象。Rust 宏采取了不同的思路，类似于 Scheme 的 syntax-rules。相比于 C++ 宏，Rust 宏可以更好地与语言的其他组件整合，因此不容易出错。宏调用始终都会以一个感叹号来标记，因此很容易在代码中发现它们，而且也不会在想调用函数时意外调用宏。Rust 宏永远不会插入不匹配的方括号或圆括号。另外，Rust 宏自带模式匹配，所以自定义可维护且好用的宏也很容易。

本章将通过几个例子展示如何编写宏。然后深入剖析一下宏的工作原理，因为跟很多 Rust 概念很像，宏也需要深入理解才能用好。最后，我们会看一下在简单的模式匹配不够用时怎么办。

20.1　宏基础

图 20-1 展示了 assert_eq! 宏源码的几个部分。


```
macro_rules! assert_eq {
    ($left:expr , $right:expr) => ({
        match (&$left, &$right) {
            (left_val, right_val) => {
                if !(*left_val == *right_val) {
                    panic!("assertion failed: `(left == right)` \
                            (left: `{:?}`, right: `{:?}`)",
                            left_val, right_val)
                }
            }
        }
    });
}
```

图 20-1：assert_eq! 宏

macro_rules! 是 Rust 中定义宏的主要方式。注意在宏定义中，assert_eq 后面没有感叹号 !。只有调用宏的时候才需要包含感叹号 !，定义的时候则不需要。

并非所有的宏都是以这种方式定义的。有几个宏，比如 file!、line! 和 macro_rules!，本身是内置在编译器中的。本章在最后还会介绍另一种方式，叫过程宏（procedural macro）。但我们会主要讨论 macro_rules!，这是（目前）编写自定义宏的最简单方式。

使用 macro_rules! 定义的宏完全基于模式匹配实现逻辑。宏的主体就是一系列规则：

```
( 模式1 ) => ( 模板1 );
( 模式2 ) => ( 模板2 );
...
```

图 20-1 所示的 assert_eq! 宏只包含一个模式和一个模板。

顺便说一下，模式和模板周围也可以不用圆括号，而用方括号或花括号。对 Rust 来说，使用哪种括号没有区别。类似地，在调用宏时，以下形式也是等价的：

```
assert_eq!(gcd(6, 10), 2);
assert_eq![gcd(6, 10), 2];
assert_eq!{gcd(6, 10), 2}
```

唯一的区别是在使用花括号时，最后的分号是可选的。按照惯例，在调用 assert_eq! 时使用圆括号，在调用 vec! 时使用方括号，而在调用 macro_rules! 时使用花括号。不过这些都只是约定而已。

20.1.1　宏扩展基础

Rust 会在编译的早期扩展宏。编译器会从头到尾读取你的源代码，同时定义并扩展遇到的宏。不能在定义宏之前调用宏，因为 Rust 对每个宏调用是边分析边扩展的。（相对而言，函数及其他特性项则不必在意顺序。比如可以先调用一个函数，然后再在后面通过导入包给出定义。）

Rust 在扩展 assert_eq! 宏调用时，过程非常类似求值 match 表达式。Rust 首先用参数去匹配模式，如图 20-2 所示。

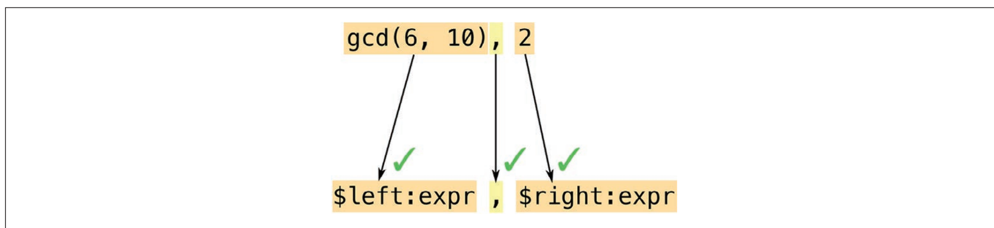


图 20-2：扩展宏的第一部分：对参数进行模式匹配

宏模式是 Rust 中的一个迷你语言。它们本质上是用于匹配代码的正则表达式。只不过正则表达式操作的是字符，而宏模式操作的是**记号**（token），比如数字、名称、标点等 Rust 程序的语法符号。这意味着在宏模式中可以随意使用注释和空白，以便让模式更容易理解。注释和空白不是记号，因此不影响匹配。

正则表达式与宏模式的另一个重要区别是圆括号、方括号和花括号在 Rust 中始终都是成对出现的。Rust 在扩展宏之前会先检查这个规则，且不限于宏模式，而是会检查这门语言的所有代码。

在这个例子中，我们的模式包含 $left:expr，它会告诉 Rust 匹配一个表达式（在这里就是 gcd(6, 10)）并将其赋值给 $left。然后 Rust 会匹配模式中的逗号与 gcd 参数后面的逗号。跟

正则表达式一样，宏模式中只有少数特殊字符会触发特殊的匹配行为；其他字符，比如逗号，则需要按字面匹配，否则匹配就会失败。最后，Rust 匹配表达式 2 并将其赋值给 `$right`。

这个模式匹配的两段代码都是 `expr` 类型，意味着它们期待表达式。20.4.1 节将介绍其他代码片段类型。

由于这个模式匹配了全部参数，因此 Rust 会扩展对应的**模板**，如图 20-3 所示。

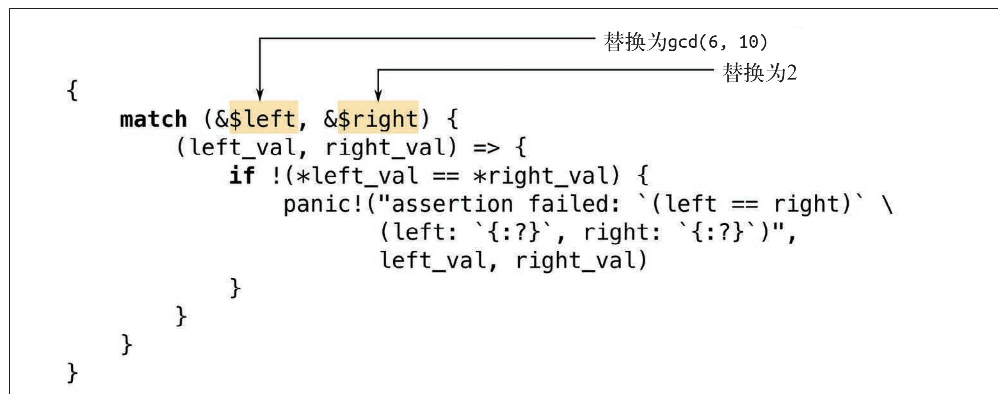


图 20-3：扩展宏的第二部分：填充模板

Rust 会将 `$left` 和 `$right` 替换为它在匹配期间找到的代码片段。

在输出模板中包含片段类型是一个常见的错误，比如在这里不写 `$left` 而是写 `$left:expr`。Rust 不会立即检测这种错误。此时，它会将 `$left` 看成一个替代结果，然后将 `:expr` 当作模板中的其他内容（即宏输出中包含的记号）看待。因此这些错误在实际**调用**宏之前不会发生，而是会生成无法编译的无效输出。如果在使用新宏后看到类似 `expected type, found `:`` 这样的错误消息，那可以查一下是不是存在这种错误。（20.3 节对类似这种情况给出了更通用的建议。）

宏模板与 Web 编程中可以使用的任何模板语言没有太大区别。唯一的区别，也是最重要的一个区别，就是宏模板输出的是 Rust 代码。

20.1.2　意外结果

把代码片段插入模板中与插入使用值的常规代码中有一些微妙的差异。这些差异一开始并不十分明显。我们看到的 `assert_eq!` 宏的模式中包含一些奇怪的代码，那是因为这些代码是宏编程代码。下面来看两个特别奇怪的地方。

首先，为什么这个宏会创建变量 `left_val` 和 `right_val`？难道把模板简化为下面这样会有什么问题吗？

```
if !($left == $right) {
    panic!("assertion failed: `(left == right)` \
            (left: `{:?}`, right: `{:?}`)", $left, $right)
}
```

要回答这个问题，可以在头脑中想象一下扩展宏调用 assert_eq!(letters.pop(), Some('z'))。输出会是什么呢？自然地，Rust 会把匹配的表达式插入模板中的多个地方。而在构建错误消息时，再把表达式全部求一遍值似乎是没必要的。不仅因为要多花一倍时间，而且第二次求值时 letters.pop() 会再从向量中移除一个值，从而导致生成不同的值！这就是真实的宏只会计算 $left 和 $right 一次，并存储它们的值的原因。

再看第二个问题：为什么这个宏要借用 $left 和 $right 值的引用？为什么不像下面这样把它们存储在变量中？

```
macro_rules! bad_assert_eq {
    ($left:expr, $right:expr) => ({
        match ($left, $right) {
            (left_val, right_val) => {
                if !(left_val == right_val) {
                    panic!("assertion failed" /* ... */);
                }
            }
        }
    });
}
```

对于眼前的这个例子来讲，宏的参数是整数，这样没问题。但如果调用者给 $left 或 $right 传入了一个 String 变量，那上面的代码就会把值从变量中转移出来！

```
fn main() {
    let s = "a rose".to_string();
    bad_assert_eq!(s, "a rose");
    println!("confirmed: {} is a rose", s); // 错误：使用转移的值“s”
}
```

因为不想让断言转移值，所以宏定义中就要借用其引用。

（你可能想知道，为什么这个宏使用 match 而不是 let 定义变量。我们也很纳闷。好像这样做并没有什么特殊的原因。使用 let 结果应该是一样的。）

简言之，宏可能导致意外结果。如果你写的宏会导致奇怪的事件发生，那就得好好追究一下这个宏的责任了。

你**不会**看到下面这个经典的 C++ 宏 bug：

```
// 给某个数值加1的C++宏；有bug
#define ADD_ONE(n) n + 1
```

因为大多数 C++ 程序员对此很熟悉，所以没必要在这里详细解释。只是告诉大家在 C++ 中，像 ADD_ONE(1) * 10 或 ADD_ONE(1 << 4) 这样并不起眼的宏调用代码可能导致令人震惊的结果。要修复这个问题，需要在宏定义中多加一层圆括号。这在 Rust 中不是必要的，因为 Rust 宏跟语言集成得更流畅。Rust 知道自己什么时候在处理表达式，因此会在把一个表达式粘贴到另一个表达式中时加上圆括号。

20.1.3 重复

标准的 vec! 宏有两种形式：

```
// 重复一个值N次
let buffer = vec![0_u8; 1000];

// 一个由逗号分隔的值列表
let numbers = vec!["udon", "ramen", "soba"];
```

可以这样来实现它：

```
macro_rules! vec {
    ($elem:expr ; $n:expr) => {
        ::std::vec::from_elem($elem, $n)
    };
    ( $( $x:expr ),* ) => {
        <[_]>::into_vec(Box::new([ $( $x ),* ]))
    };
    ( $( $x:expr ),+ ,) => {
        vec![ $( $x ),* ]
    };
}
```

这里定义了 3 条规则。我们先来解释多条规则的工作方式，然后再逐一看每条规则。

Rust 在扩展类似 `vec![1, 2, 3]` 这样的宏调用时，首先会用参数 `1, 2, 3` 去匹配第一条规则，也就是 `$elem:expr ; $n:expr`。这次匹配会失败：`1` 是表达式，但模式要求接下来是一个分号，可参数里没有。因此 Rust 会转移到第二条规则，以此类推。如果没有规则匹配，就是一个错误。

第一条规则处理类似 `vec![0u8; 1000]` 这样的用法。恰好有一个标准的函数满足要求，即 `std::vec::from_elem`，因此这条规则的模板很好理解。

第二条规则处理 `vec!["udon", "ramen", "soba"]` 这样的调用。模式 `$( $x:expr ),*` 使用了之前没有见过的一个特性：重复。它匹配 0 或多个表达式，以逗号分隔。更概括地说，就是语法 `$( PATTERN ),*` 用于匹配任何以逗号分隔的列表，其中列表的每一项都匹配 `PATTERN`。

这里的 `*` 与其在正则表达式中的含义相同（表示“匹配 0 次或多次”），当然正则表达式中是没有 `,*` 这种表示重复的特殊语法的。在这里也可以使用 `+` 表示至少匹配一次。不过没有 `?` 语法。表 20-1 总结了所有的重复模式表示法。

表20-1：重复的模式

模　式	含　义
`$( ... )*`	匹配 0 或多次，没有分隔符
`$( ... ),*`	匹配 0 或多次，以逗号分隔
`$( ... );*`	匹配 0 或多次，以分号分隔
`$( ... )+`	匹配 1 或多次，没有分隔符
`$( ... ),+`	匹配 1 或多次，以逗号分隔
`$( ... );+`	匹配 1 或多次，以分号分隔

码片段 `$x` 不只是一个表达式，而是一组表达式。针对这条规则的模板也使用了重复语法：

```
<[_]>::into_vec(Box::new([ $( $x ),* ]))
```

同样，这种情况也有标准方法可以实现我们想要的。以上代码创建了一个装箱数组，然后使用 `[T]::into_vec` 方法将装箱数组转换为了向量。

开头的 `<[_]>` 表示“某种类型值的切片”，但并没有明确的类型，而是希望 Rust 推断元素类型。名字是纯标识符的类型可以在表达式中不经任何装修而直接使用，`fn()`、`&str` 或 `[_]` 等类型则必须包装在一对尖括号中。

重复出现在这个模板的最后，也就是 `$($x),*` 这部分。这跟 `$( ... ),*` 模式的语法相同，表示迭代由 `$x` 匹配的一组表达式，并将它们全部插入模板中，以逗号分隔。

在这里，重复的输出看起来跟输入一样。不过，也不一定非要如此。比如，也可以这样实现：

```
( $( $x:expr ),* ) => {
    {
        let mut v = Vec::new();
        $( v.push($x); )*
        v
    }
};
```

`$( v.push($x); )*` 对 `$x` 中的每个表达式都插入了一次对 `v.push()` 的调用。

与 Rust 的其他特性不同，使用 `$( ... ),*` 的模式不会自动支持末尾的逗号可选。不过，对此我们有一个标准的解决方案，即通过再添加一条规则来支持末尾的逗号。这就是 `vec!` 宏的第三条规则的用途：

```
( $( $x:expr ),+ ,) => {  // 如果末尾存在逗号，去掉它再重试
    vec![ $( $x ),* ]
};
```

这里使用 `$( ... ),+ ,` 来匹配带有额外逗号的列表。然后在模板中，再递归调用 `vec!`，把逗号去掉。这次就会匹配第二条规则了。

20.2 内置宏

Rust 编译器提供了几个内置的宏，以辅助我们编写自定义的宏。这些宏都不能仅凭使用 `macro_rules!` 来实现。它们是硬编码在 `rustc` 中的。

- **`file!()`** 扩展为一个字符串字面量，即当前文件名。**`line!()`** 和 **`column!()`** 扩展为一个 `u32` 字面量，代表当前行（从 1 开始）和列（从 0 开始）。

  如果一个宏调用了另一个宏，另一个宏又调用了其他宏，而且它们都在不同的文件中，那么最后一次对 `file!()`、`line!()` 或 `column!()` 的调用会扩展为表示**第一次**宏调用的位置。

- **`stringify!(...tokens...)`** 扩展为一个字符串字面量，包含给定的记号。`assert!` 宏使用这个内置宏生成包含断言代码的错误消息。

  参数中的宏调用**不会**扩展，即 `stringify!(line!())` 会扩展为字符串 `"line!()"`。

Rust 从记号开始构建这个字符串，因此字符串中不包含换行或注释。

- **`concat!(str0, str1, ...)`** 扩展为一个字符串字面量，是拼接其参数之后的结果。

Rust 还定义了以下查询构建环境的宏。

- **`cfg!(...)`** 扩展为一个布尔值常量，如果当前构建配置与括号中的条件匹配则为 `true`。例如，`cfg!(debug_assertions)` 在编译时启用调试断言的条件下为 `true`。

  这个宏支持与 8.5 节介绍的 `\#[cfg(...)]` 属性相同的语法，但它不是条件编译，而是返回 `true` 或 `false`。

- **`env!("VAR_NAME")`** 扩展为一个字符串，即指定环境变量在编译时的值。如果指定的变量不存在，就是一个编译错误。

  除非 Cargo 在编译一个包时设置了有意思的环境变量，否则这个宏没什么用。例如，要取得包的当前版本字符串，可以这样写：

  ```
  let version = env!("CARGO_PKG_VERSION");
  ```

  关于所有这些环境变量的详细信息，请参考 Cargo 的文档。

- **`option_env!("VAR_NAME")`** 与 `env!` 相同，只不过其返回 `Option<&'static str>`，如果指定变量没有设置，则返回 `None`。

以下 3 个内置的宏支持从另一个文件取得代码或数据。

- **`include!("file.rs")`** 扩展为指定文件的内容，必须是有效的 Rust 代码，比如表达式或特性项的序列。
- **`include_str!("file.txt")`** 扩展为一个 `&'static str`，包含指定文件的文本。可以像下面这样使用它。

  ```
  const COMPOSITOR_SHADER: &str =
      include_str!("../resources/compositor.glsl");
  ```

  如果指定的文件不存在，或者文本不是有效 UTF-8，就会导致编译错误。

- **`include_bytes!("file.dat")`** 也一样，只不过是将文件作为二进制数据而非 UTF-8 文本来对待。结果是 `&'static [u8]`。

与所有宏一样，这些内置的宏都是在编译时被处理的。如果文件不存在或者不能读，那编译就会失败。所有宏都不能在运行时失败。在任何情况下，如果文件名是相对路径，都会相对于包含当前文件的目录来解析。

20.3 调试宏

调试任性的宏会很有挑战性。最大的问题是宏扩展的过程是不可见的。Rust 经常会扩展所有宏，在发现某些错误时打印出一条错误消息，但不会显示包含错误的完全扩展后的代码。

下面介绍 3 个有助于排除宏错误的工具。（这些特性全部是不稳定的，但因为它们实际上只用于开发阶段，而不是要提交的代码中，所以实践中并不是什么大问题。）

第一个，也是最简单的，可以使用 rustc 展示代码在扩展所有宏之后的样子。使用 cargo build --verbose 可以看到 Cargo 是如何调用 rustc 的。复制 rustc 命令行，然后加上 -Z unstable-options --pretty expanded 选项。完全扩展后的代码会显示在终端上。可惜的是，只有在你的代码没有语法错误时，这种方式才可以使用。

第二个，Rust 提供了一个 log_syntax!() 宏，其可以在编译时把它的参数打印到终端上。可以使用它实现类似 println! 的调试。这个宏要求 #![feature(log_syntax)] 特性标志。

第三个，可以让 Rust 编译器把所有宏调用的日志打印到终端上。在你代码的某个地方插入 trace_macros!(true);。然后，Rust 每扩展一个宏，都会打印宏的名字和参数。例如，以下程序：

```
#![feature(trace_macros)]

fn main() {
    trace_macros!(true);
    let numbers = vec![1, 2, 3];
    trace_macros!(false);
    println!("total: {}", numbers.iter().sum::<u64>());
}
```

会输出以下结果：

```
$ rustup override set nightly
...
$ rustc trace_example.rs
note: trace_macro
 --> trace_example.rs:5:19
  |
5 |     let numbers = vec![1, 2, 3];
  |                   ^^^^^^^^^^^^^
  |
  = note: expanding `vec! { 1 , 2 , 3 }`
  = note: to `< [ _ ] > :: into_vec ( box [ 1 , 2 , 3 ] )`
```

编译器会显示每次宏调用的代码，既包括扩展前的也包括扩展后的。而 trace_macros!(false); 又关闭了宏追踪，因此就不会追踪对 println!() 的调用了。

20.4 json!宏

前面已经讨论了 macro_rules! 的核心特性。接下来，本节会循序渐进地开发一个构建 JSON 数据的宏。通过这个例子，可以了解怎么开发宏，同时本节也介绍了 macro_rules! 剩下的几个特性。此外，我们也会对如何确保你的宏如期运转给出一些建议。

第 10 章曾经展示过一个用枚举表示的 JSON 数据：

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
```

```
        String(String),
        Array(Vec<Json>),
        Object(Box<HashMap<String, Json>>)
    }
```

可是用来创建这个 Json 值的代码显得非常啰唆：

```
    let students = Json::Array(vec![
        Json::Object(Box::new(vec![
            ("name".to_string(), Json::String("Jim Blandy".to_string())),
            ("class_of".to_string(), Json::Number(1926.0)),
            ("major".to_string(), Json::String("Tibetan throat singing".to_string()))
        ].into_iter().collect())),
        Json::Object(Box::new(vec![
            ("name".to_string(), Json::String("Jason Orendorff".to_string())),
            ("class_of".to_string(), Json::Number(1702.0)),
            ("major".to_string(), Json::String("Knots".to_string()))
        ].into_iter().collect()))
    ]);
```

我们希望可以使用像下面这样 JSON 风格的语法：

```
    let students = json!([
        {
            "name": "Jim Blandy",
            "class_of": 1926,
            "major": "Tibetan throat singing"
        },
        {
            "name": "Jason Orendorff",
            "class_of": 1702,
            "major": "Knots"
        }
    ]);
```

其实就是需要开发一个 json! 宏，它接收一个 JSON 值作为参数，然后将其扩展为类似前面代码中的 Rust 表达式。

20.4.1 片段类型

要开发一个复杂的宏，第一项工作是规划好如何匹配（**解析**）期望的输入。

我们已经知道这个宏应该包含多条规则，因为 JSON 数据可以包含多种数据类型，如对象、数组、数值，等等。事实上，可以假设对每种 JSON 类型都写一条规则：

```
    macro_rules! json {
        (null)    => { Json::Null };
        ([ ... ]) => { Json::Array(...) };
        ({ ... }) => { Json::Object(...) };
        (???)     => { Json::Boolean(...) };
        (???)     => { Json::Number(...) };
        (???)     => { Json::String(...) };
    }
```

这样肯定不行，因为宏模式并没有提供表达最后 3 种形式的方式，稍后我们会再讨论。先来看前 3 种情况，至少它们有差异明显的记号，所以先从它们入手。

第一条规则这样写就可以了：

```
macro_rules! json {
    (null) => {
        Json::Null
    }
}

#[test]
fn json_null() {
    assert_eq!(json!(null), Json::Null); // 通过！
}
```

要支持 JSON 数组，可以尝试用 expr 来匹配元素：

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:expr ),* ]) => {
        Json::Array(vec![ $( $element ),* ])
    };
}
```

可惜这样并不能匹配所有 JSON 数组。下面的测试说明了问题：

```
#[test]
fn json_array_with_json_element() {
    let macro_generated_value = json!(
        [
            // 有效的JSON，但不匹配$element:expr
            {
                "pitch": 440.0
            }
        ]
    );
    let hand_coded_value =
        Json::Array(vec![
            Json::Object(Box::new(vec![
                ("pitch".to_string(), Json::Number(440.0))
            ].into_iter().collect()))
        ]);
    assert_eq!(macro_generated_value, hand_coded_value);
}
```

模式 `$( $element:expr ),*` 的意思是“逗号分隔的 Rust 表达式的列表”。但很多 JSON 值，特别是对象，并不是有效的 Rust 表达式。所以不会匹配。

因为我们想要匹配的代码并非都是表达式，所以 Rust 还支持了其他一些片段类型，如表 20-2 所示。

表20-2：macro_rules!宏支持的片段类型

<table>
<tr><th>片段类型</th><th>匹配（示例）</th><th>后面可以跟……</th></tr>
<tr><td>expr</td><td>表达式：
2 + 2，"udon"，x.len()</td><td>=> , ;</td></tr>
<tr><td>stmt</td><td>表达式或声明，不包含末尾的分号
（很难用，优先使用 expr 或 block）</td><td>=> , ;</td></tr>
<tr><td>ty</td><td>类型：
String、Vec<u8>、(&str, bool)</td><td>=> , ; = | { [: >
as where</td></tr>
<tr><td>path</td><td>路径（8.2.2 节讨论过）：
ferns、::std::sync::mpsc</td><td>=> , ; = | { [: >
as where</td></tr>
<tr><td>pat</td><td>模式（10.2 节讨论过）：
_、Some(ref x)</td><td>=> , = | if in</td></tr>
<tr><td>item</td><td>特性项（6.3 节讨论过）：
struct Point { x: f64, y: f64}、mod ferns;</td><td>不限</td></tr>
<tr><td>block</td><td>代码块（6.2 节讨论过）：
{ s += "ok\n"; true }</td><td>不限</td></tr>
<tr><td>meta</td><td>属性体（8.5 节讨论过）：
inline、derive(Copy, Clone)、doc="3D models."</td><td>不限</td></tr>
<tr><td>ident</td><td>标识符：
std、Json、longish_variable_name</td><td>不限</td></tr>
<tr><td>tt</td><td>记号树（参见正文）：
;、>=、{}、[0 1 (+ 0 1)]</td><td>不限</td></tr>
</table>

表中所列的大多数选项严格对应 Rust 语法。比如，expr 类型只匹配 Rust 表达式（而非 JSON 值），ty 匹配 Rust 类型，等等。这些选项都不能扩展，也就是说没有办法定义可以让 expr 认可的新的算术操作符或新关键字。我们也不能使用其中任何一项来匹配任意 JSON 数据。

最后两个片段类型，即 ident 和 tt，支持匹配看起来不是 Rust 代码的宏参数，其中 ident 匹配任意标识符，tt 匹配一个**记号树**（token tree）。所谓记号树，可以是一个配对正确的括号，如 (...)、[...] 或 {...}，以及括号中的任何东西，包括嵌套的记号树，也可以是一个单独的非括号记号，如 1926 或 "Knots"。

记号树正是实现 json! 宏所需要的。每个 JSON 值都是一个记号树，比如数值、字符串、布尔值，以及 null 都是单独的记号，对象和数组则是有括号的记号。因此，可以把模式改写成下面这样：

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        Json::Array(...)
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        Json::Object(...)
    };
    ($other:tt) => {
```

```
            ... // TODO：返回Number、String或Boolean
        };
    }
```

这个版本的 json! 宏可以匹配所有 JSON 数据。接下来只要生成正确的 Rust 代码即可。

为确保将来无论增加什么语法特性都不会破坏你今天写的宏，Rust 对模式中可以跟在片段后面的记号做了限制。表 20-1 中的“后面可跟……”那一列给出了允许的记号。例如，模式 `$x:expr ~ $y:expr` 是错误的，因为 ~ 不允许出现在 expr 后面。模式 `$vars:path : $t:ty` 则没有问题，因为 `$vars:path` 后面跟着一个冒号，而冒号是允许出现在 path 后面的；`$t:ty` 后面什么也没有，这当然没有问题。

20.4.2 在宏里使用递归

前面已经展示过一个在宏中自我调用的简单例子：vec! 宏的实现使用递归支持了末尾的分号。下面来看一个更重要的例子：json! 也需要递归调用自己。

可以先试试不使用递归来支持 JSON 数组，比如：

```
([ $( $element:tt ),* ]) => {
    Json::Array(vec![ $( $element ),* ])
};
```

但这样不行。这样会把 JSON 数据（`$element` 记号树）直接粘贴到一个 Rust 表达式中。但它们是两种不同的语言。

为此，需要把数组中的每个元素都从 JSON 转换为 Rust。所幸的是，有一个宏正是做这个用的，它就是我们正在写的这个！

```
([ $($element:tt),* ]) => {
    Json::Array(vec![ $( json!($element) ),* ])
};
```

对象也可以采用同样的方式获得支持：

```
({ $($key:tt : $value:tt),* }) => {
    Json::Object(Box::new(vec![
        $( ($key.to_string(), json!($value)) ),*
    ].into_iter().collect()))
};
```

编译器对宏施加了一个递归次数的限制，默认为 64 次调用。这对于 json! 常规使用已经足够了，但特别复杂情况下的递归有可能会碰到这个限制。为此，可以在使用这个宏的包顶部添加如下属性：

```
#![recursion_limit = "256"]
```

我们的 json! 宏差不多要完成了。剩下的工作是支持布尔值、数值和字符串值。

20.4.3 在宏里使用特型

编写复杂的宏总会遇到难题。重要的是要记住，宏本身并非解决验证唯一的工具。

这里，我们需要支持 json!(true)、json!(1.0) 和 json!("yes")，将这些（涵盖各种可能性的）值转换为适当的 Json 值。但宏并不擅长区分类型。可以想象如下实现：

```
macro_rules! json {
    (true) => {
        Json::Boolean(true)
    };
    (false) => {
        Json::Boolean(false)
    };
    ...
}
```

这种方式一看就行不通。布尔值只有两种可能性，但数值呢？更不用说字符串了。

好在我们有一个把不同类型值转换为某种指定类型的标准方式：13.9 节介绍的 From 特型。只需为几个类型实现这个特型：

```
impl From<bool> for Json {
    fn from(b: bool) -> Json {
        Json::Boolean(b)
    }
}

impl From<i32> for Json {
    fn from(i: i32) -> Json {
        Json::Number(i as f64)
    }
}

impl From<String> for Json {
    fn from(s: String) -> Json {
        Json::String(s)
    }
}

impl<'a> From<&'a str> for Json {
    fn from(s: &'a str) -> Json {
        Json::String(s.to_string())
    }
}
...
```

事实上，全部 12 种数值类型的实现都非常相似。因此有必要专门为此写一个宏，以避免冗余的复制 – 粘贴：

```
macro_rules! impl_from_num_for_json {
    ( $( $t:ident )* ) => {
        $(
            impl From<$t> for Json {
                fn from(n: $t) -> Json {
                    Json::Number(n as f64)
                }
            }
```

```
        )*
    };
}

impl_from_num_for_json!(u8 i8 u16 i16 u32 i32 u64 i64 usize isize f32 f64);
```

这样就可以使用 `Json::from(value)` 将任何支持类型的值转换为 `Json` 了。在我们的宏里，可以这样使用：

```
($other:tt) => {
    Json::from($other) // 处理布尔值、数值和字符串
};
```

在我们的 `json!` 宏中加上这条规则，让它通过目前为止所有的测试。所有代码合在一起，就是这个样子：

```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $($element:tt),* ]) => {
        Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $($key:tt : $value:tt),* }) => {
        Json::Object(Box::new(vec![
            $( ($key.to_string(), json!($value)) ),*
        ].into_iter().collect()))
    };
    ($other:tt) => {
        Json::from($other)  // 处理布尔值、数值和字符串
    };
}
```

结果，这个宏意外地支持在 JSON 数据中使用变量，甚至任意 Rust 表达式，这是个方便的额外特性：

```
let width = 4.0;
let desc =
    json!({
        "width": width,
        "height": (width * 9.0 / 4.0)
    });
```

因为 `(width * 9.0 / 4.0)` 加了圆括号，所以它是一个单独的记号树，结果这个宏在解析对象时用 `$value:tt` 成功匹配到了它。

20.4.4 作用域与自净宏

编写宏的一个比较隐晦的难题是它涉及把不同作用域的代码粘贴到一起。接下来我们会介绍 Rust 处理作用域的两种方式，一种针对局部变量和参数，另一种针对其他情况。

为说明这个问题的重要性，下面先重写一下解析 JSON 对象的规则（前面展示的 `json!` 宏的第三个规则），以避免使用临时向量。可以这样来写：

```
({ $($key:tt : $value:tt),* }) => {
    {
        let mut fields = Box::new(HashMap::new());
        $( fields.insert($key.to_string(), json!($value)); )*
        Json::Object(fields)
    }
};
```

现在，填充 HashMap 是通过重复调用 .insert() 方法而不是使用 collect() 来实现的。这意味着需要把这个映射保存在一个临时变量，也就是这里的 fields 中。

但是如果调用 json! 的代码恰好也使用了自己所拥有的一个变量，而且这个变量也叫 fields 该怎么办呢？

```
let fields = "Fields, W.C.";
let role = json!({
    "name": "Larson E. Whipsnade",
    "actor": fields
});
```

扩展这个宏会把两块代码粘贴在一起，两边都使用了名字 fields，但保存的数据不同！

```
let fields = "Fields, W.C.";
let role = {
    let mut fields = Box::new(HashMap::new());
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
    fields.insert("actor".to_string(), Json::from(fields));
    Json::Object(fields)
};
```

看起来只要宏使用了临时变量，这似乎就是一个无法避免的陷阱，你可能已经在思考可能的解决方案了。或许应该把 json! 宏里使用的变量重命名为一个调用者不太可能传进来的名字，比如不叫 fields，而叫 __json$fields。

令人惊喜的是**宏就是这样做的**。Rust 会这样为我们重新命名变量！这个特性最初是在 Scheme 宏里实现的，叫作**自净**（hygiene）。所以 Rust 也说自己有**自净宏**。

理解宏自净的最简单方式是在宏每次被扩展时，都会给来自宏自身的这部分代码染上不同的颜色。

这样，不同颜色的变量就被当成不同的名字来对待了：

```
let fields = "Fields, W.C.";
let role = {
    let mut fields = Box::new(HashMap::new());
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
    fields.insert("actor".to_string(), Json::from(fields));
    Json::Object(fields)
};
```

注意，由宏调用者粘贴进来且会粘贴到输出的代码，比如 "name" 和 "actor"，仍会保持其原先的颜色（黑色）。只有源自宏模板的记号才被染了色。

现在就有了一个（在调用者中声明的）名为 fields 的变量和另一个（由宏引入的）名为

fields 的变量。因为两个名字的颜色不同，所以它们不会被混淆。

如果宏真的需要引用调用者作用域中的变量，调用者就必须把这个变量的名字传给宏。

（染色的比喻并不代表它就是自净的真实工作方式。真正的机制比这还要稍微智能一些。如果两个标识符相同，不管什么“颜色”，只有它们都指向宏及其调用者所在作用域的公共变量，才会被认为是相同的。但这种情况在 Rust 中很少见。如果你理解了前面的例子，那么对自净宏的认识就够了。）

你可能注意到了，在这个宏被扩展时，其他几个标识符被染上了不止一种颜色，比如 Box、HashMap 和 Json 的颜色就不一样。虽然颜色不同，但 Rust 识别这些类型名并非难事，因为 Rust 中的自净只限于局部变量和参数。如果遇到常量、类型、方法、模块和宏名字，那 Rust 就是“色盲”了。

这意味着如果一个模块要使用我们的 json! 宏，而 Box、HashMap 或 Json 并不在这个模块的作用域中，那宏就不会起作用。下一节会介绍如何避免这个问题。

首先，我们要知道 Rust 的严格自净会带来阻碍，需要先解决这个问题。假设有很多函数要包含下面这行代码：

```
let req = ServerRequest::new(server_socket.session());
```

复制粘贴这行代码是件痛苦的事。能不能写个宏来代替呢？

```
macro_rules! setup_req {
    () => {
        let req = ServerRequest::new(server_socket.session());
    }
}

fn handle_http_request(server_socket: &ServerSocket) {
    setup_req!(); // 声明req，使用server_socket
    ... // 使用req的代码
}
```

像这样写是不行的。它需要宏里的 server_socket 引用在函数中声明的局部变量 server_socket，对变量 req 则正好相反。但自净阻止了宏里的名字与其他作用域中名字的“冲突”。即使在这种情况下也是一样，而这种“冲突”正是我们想要的。

解决方案是给宏传入你想同时在宏代码内部和外部使用的标识符：

```
macro_rules! setup_req {
    ($req:ident, $server_socket:ident) => {
        let $req = ServerRequest::new($server_socket.session());
    }
}

fn handle_http_request(server_socket: &ServerSocket) {
    setup_req!(req, server_socket);
    ... // 使用req的代码
}
```

因为此时的 req 和 server_socket 是由函数提供的，所以它们的“颜色”自然就代表那个作用域。

自净使得这个宏使用起来有点烦琐，不过这是一个特性，而不是 bug。很容易推断，自净宏知道不能背着你干扰局部变量。如果在函数中搜索某个标识符，比如 server_socket，你会发现使用它的所有地方，包括宏调用。

20.4.5　导入和导出宏

由于宏是在编译早期被扩展的，因此在 Rust 知道你项目的整个模块结构之前，它们不能以常规方式导入和导出。

在只有一个包的情况下：

- 在一个模块中可见的宏会自动在其子模块中可见；
- 要从一个模块中“向上”把宏导出到其父模块，需要使用 #[macro_use] 属性。例如，假设我们的 lib.rs 包含如下代码：

  ```
  #[macro_use] mod macros;
  mod client;
  mod server;
  ```

  所有在 macros 模块中定义的宏都会导入 lib.rs 中，因此对这个包的其他部分也是可见的，包括 client 和 server。

在使用多个包的情况下：

- 要从另一个包导入宏，需要在 extern crate 声明上使用 #[macro_use]；
- 要从一个包中导出宏，需要将每个要公开的宏都标记为 #[macro_export]。

当然，添加上述任何属性或标记，都意味着你的宏可能会被其他模块调用。为此，导出的宏不应该依赖于作用域中存在某种东西，因为谁也不知道使用它的作用域中会有什么。就算标准前置模块中的特性仍然有被遮蔽的可能。

相反，对外的宏应该使用绝对路径指向自己使用的任何名字。macro_rules! 以一个特殊的片段 $crate 为此提供便利。这个片段类似于宏定义所在包的根模块的绝对路径。比如，不要使用 Json，而是写成 $crate::JSon。这样即使没有导入 Json，你的宏也可以照样使用。HashMap 也可以被修改为 ::std::collections::HashMap 或 $crate::macros::HashMap。对后一种写法，还必须再导出 HashMap，因为 $crate 不能用于访问包的私有特性。它实际上会被扩展为类似 ::jsonlib 这样的形式，这是一个普通路径。可见性规则并不会受影响。

当把宏转移到它自己的模块 macros 中并修改为使用 $crate 之后，它就变成了下面这样。这是最终的版本。

```
// macros.rs
pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
    (null) => {
        $crate::Json::Null
```

```
        };
        ([ $( $element:tt ),* ]) => {
            $crate::Json::Array(vec![ $( json!($element) ),* ])
        };
        ({ $( $key:tt : $value:tt ),* }) => {
            {
                let mut fields = $crate::macros::Box::new(
                    $crate::macros::HashMap::new());
                $( fields.insert($crate::ToString::to_string($key), json!($value)); )*
                $crate::Json::Object(fields)
            }
        };
        ($other:tt) => {
            $crate::Json::from($other)
        };
    }
```

由于 `.to_string()` 方法源自标准的 `ToString` 特型，因此这里也使用了 `$crate` 来引用它，所用语法是 11.3 节介绍过的完全限定的方法调用形式：`$crate::ToString::to_string($key)`。对这个宏而言，并不是必须这样引用才能保证其工作，因为 `ToString` 位于标准前置模块。不过，如果你的宏用到了一个调用它时可能并不在作用域内的特型方法，那么完全限定的方法调用是最佳选择。

20.5　匹配时避免语法错误

下面的宏看似合理，但会给 Rust 制造麻烦：

```
    macro_rules! complain {
        ($msg:expr) => {
            println!("Complaint filed: {}", $msg);
        };
        (user : $userid:tt , $msg:expr) => {
            println!("Complaint from user {}: {}", $userid, $msg);
        };
    }
```

假设这样调用它：

```
    complain!(user: "jimb", "the AI lab's chatbots keep picking on me");
```

肉眼可见，这明显匹配第二个模式。但 Rust 会先尝试匹配第一个模式，企图用 `$msg:expr` 来匹配所有输入。对我们而言，这就是问题的根源。当然，`user: "jimb"` 不是表达式，因此我们会得到一个语法错误。Rust 拒绝掩盖语法错误，毕竟宏已经很难调试了。相反，它会立即报告这个错误，停止编译。

如果是模式中的记号匹配失败，Rust 就会切换到下一条规则。只有语法错误是致命的，而且只在匹配片段的时候才会发生。

这里的问题并不难理解：我们是在一个错误的规则中企图匹配片段 `$msg:expr`。结果并不匹配，因为我们甚至都不应该出现在这里。调用者想匹配的是其他规则。有两个简单的方法能避免这个问题。

第一个方法是避免容易混淆的规则。比如，可以改一改这个宏，让它的每个模式都以一个不同的标识符开头：

```
macro_rules! complain {
    (msg : $msg:expr) => {
        println!("Complaint filed: {}", $msg);
    };
    (user : $userid:tt , msg : $msg:expr) => {
        println!("Complaint from user {}: {}", $userid, $msg);
    };
}
```

如果宏参数以 `msg` 开头，就匹配规则一，如果以 `user` 开头，就匹配规则二。无论参数是什么，我们都会在匹配片段之前就知道正确的规则。

另一个避免这种欺骗性语法错误的方法是把更具体的规则放在前面。把 `user :` 规则放到前面就可以修改 `complain!` 的问题，因为永远不会碰到会导致语法错误的规则。

20.6 超越macro_rules!

宏模式可以解析比 JSON 更复杂的输入，不过我们发现这种复杂性很快就会失控。

Daniel Keep 等人编写的 *The Little Book of Rust Macros* 是一本非常棒的高级 `macro_rules!` 编程指南。这本书讲解清晰、构思巧妙，比本章更详细地介绍了宏扩展的方方面面。它也展示了一些非常聪明的技术，比如把 `macro_rules!` 模式作为一种小众编程语言集成到服务中，去解析复杂输入。我们对这种做法并不热衷。建议大家小心使用。

Rust 1.15 引入了一种不同的机制，叫**过程宏**（procedural macro）。这个特性支持扩展 `#[derive]` 属性，以处理自定义特型，如图 20-4 所示。

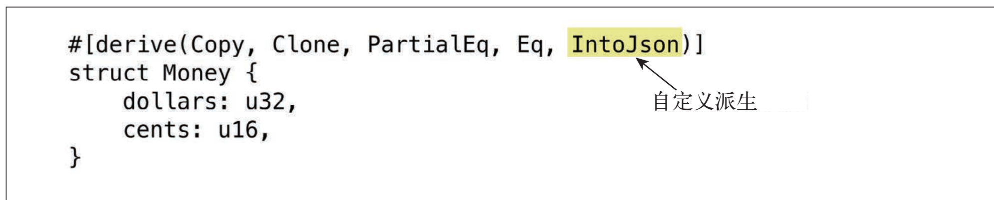


图 20-4：通过 `#[derive]` 属性调用假想的 `IntoJson` 过程宏

没有 `IntoJson` 特型，但没关系：过程宏可以利用这个接入点插入它想要的任何代码（在这里，可能是 `impl From<Money> for Json { ... }`）。

过程宏之所以是“过程”，主要因为它是作为 Rust 函数实现的，并非一个声明性的规则集。在写作本书时，过程宏仍然是一个新特性，还会继续发展，因此建议大家关注其在线文档。

看到这里，或许你已经不喜欢宏了。然后呢？一个替代方案是使用构建脚本生成 Rust 代码。Cargo 文档中关于构建脚本的内容展示了如何一步一步来操作。总之就是写一个程序，让它生成你想要的 Rust 代码。然后在 Cargo.toml 中加上一行，以便在构建期间同时运行该程序，并且使用 `include!` 把生成的代码包含到你的包中。

第21章

不安全代码

> 希望没有人认为我软弱、无能、消沉；
>
> 希望他们理解我的与众不同：
>
> 对敌人是威胁，对朋友很忠诚。
>
> 这样的生活无上荣耀。
>
> ——*Medea*，Euripides

系统编程不为人知的乐趣在于，在每一个安全语言和精心设计的抽象之下，都奔腾着狂放不羁的不安全机器语言和自由散漫的位。在 Rust 中也可以写出这种代码。

迄今为止，本书介绍的这门语言借助类型、生命期、边界检查等手段，可以完全自动化地保证你的程序没有内存错误和数据争用。但这种自动化推理有其局限性，Rust 中很多有价值的技术并不能认为是安全的。

不安全的代码让你告诉 Rust："这时候，信任我。"通过把一个块或函数标记为不安全，你可以获得调用标准库中 `unsafe` 函数、解引用不安全指针和调用以其他语言（比如 C 和 C++）编写的函数等能力。Rust 所有常规的安全检查仍然适用：类型检查、生命期检查，以及对索引的边界检查都会正常进行。不安全代码只是启用了一小部分额外的特性。

跨越到安全 Rust 边界之外的能力也是 Rust 实现自身很多最基本特性的基础，就像 C 和 C++ 系统通常所做的那样。不安全代码可以让 `Vec` 类型更高效地管理其缓冲区，让 `std::io` 模块与操作系统交谈，让 `std::thread` 和 `std::sync` 模块提供并发原语。

本章介绍了如下使用不安全特性的基础知识。

- Rust 的 `unsafe` 块用于将普通的、安全的 Rust 代码与使用不安全特性的代码隔离开。
- 可以将函数标记为 `unsafe`，提醒调用者注意遵循额外的协议，以避免未定义行为。

- 原始指针及它们的方法可以不受限制地访问内存，从而构建正常情况下 Rust 类型系统会禁止的数据结构。
- 理解未定义行为的定义可以帮你认识到为什么它意味着极为严重的后果，而不仅仅是得到错误的结果。
- Rust 的外来函数接口让你能够使用其他语言编写的库。
- 不安全特型，类似于 `unsafe` 函数，对所有实现（而非调用者）添加了必须遵守的协议。

21.1 不安全源自哪里

本书一开始就展示了一个由于没有遵守 C 标准中规则而意外崩溃的 C 程序。在 Rust 中也可以做到：

```
$ cat crash.rs
fn main() {
    let mut a: usize = 0;
    let ptr = &mut a as *mut usize;
    unsafe {
        *ptr.offset(3) = 0x7ffff72f484c;
    }
}
$ cargo build
   Compiling unsafe-samples v0.1.0
    Finished debug [unoptimized + debuginfo] target(s) in 0.44 secs
$ ../../target/debug/crash
crash: Error: .netrc file is readable by others.
crash: Remove password or make file unreadable by others.
Segmentation fault (core dumped)
$
```

这个程序借用了本地变量 `a` 的一个可修改引用，将其转型为一个原始指针类型 `*mut usize`，然后使用 `offset` 方法在内存中又产生了一个 3 个字的指针。这个地址恰好是存储 `main` 返回地址的位置。由于程序用一个常量覆盖返回地址，因此 `main` 的返回行为就变得非常奇怪。导致这次崩溃的原因是程序错误地使用了不安全特性——具体来说就是解引用原始指针的能力。

不安全特性都会附带一个**协议**（contract），也就是 Rust 不能自动强制的规则。对这个协议，你必须遵循，否则就会导致**未定义行为**（undefined behavior）。

协议并不止常规的类型检查和生命期检查，而是对特定不安全特性做了进一步约束。通常，Rust 本身并不知道有协议，因为这些协议只存在于相关特性的文档中。例如，原始指针类型有一个协议，就是禁止解引用已经超出其最初引用值边界的指针。前面例子中的表达式 `*ptr.offset(3) = ...` 违反了这个协议。不过，正如代码输出所示，Rust 毫无怨言地编译了这段程序，因为它的安全检查没有检测到这个违反协议的地方。因此在使用不安全特性时，作为程序员就要承担起检查你的代码是否遵守了它们协议的职责。

很多特性需要遵循一些规则才能正确使用，不过这些规则并不是这里所说的协议，除非违反它们的结果包含未定义行为。未定义行为是 Rust 坚决认为你的代码永远不会出现的行为。例如，Rust 认为你不会用其他东西覆盖一个函数调用的返回地址。能够通过 Rust 常

规安全检查并按照它所使用特性协议编译的代码不可能做这样的事。前面的程序违反了这个原始指针协议，其行为是未定义的，它已经脱离了正轨。

如果你的代码表现出未定义行为，那你已经破坏了跟 Rust 之间一半的约定，Rust 拒绝预测后果。从系统库和崩溃中发掘不相关的错误消息是一种可能的后果，把你计算机的控制权拱手让给攻击者则是另一种可能的后果。最终效果可能因 Rust 发布的版本而不同，没有警告。不过有时候，未定义行为未必有可见的后果。例如，如果 `main` 函数永远不返回（比如调用 `std::process::exit` 提前终止程序），那么被破坏的返回地址可能就不是问题。

应该只在 `unsafe` 块或 `unsafe` 函数中使用不安全特性，接下来几节将分别介绍它们。这样可以避免在不知不觉中使用不安全特性，因为使用不安全特性时必须先写一个 `unsafe` 块或函数。Rust 确保你一定知道了自己的代码可能要遵守额外的规则。

21.2 不安全的块

`unsafe` 块就是在普通 Rust 块前面加上 `unsafe` 关键字，然后就可以在块中使用不安全特性了：

```
unsafe {
    String::from_utf8_unchecked(ascii)
}
```

如果块前面没有 `unsafe` 关键字，那么 Rust 会反对使用 `from_utf8_unchecked`，因为它是一个 `unsafe` 函数。而有了 `unsafe` 块，这行代码你想在哪里使用就在哪里使用。

与普通 Rust 块一样，`unsafe` 块的值是它最后表达式的值，如果没有表达式就是 `()`。前面展示的对 `String::from_utf8_unchecked` 的调用提供了那个块的值。

`unsafe` 块解锁了 4 个额外选项。

- 可以调用 `unsafe` 函数。每个 `unsafe` 函数必须根据自己的意图指定自己的协议。
- 可以解引用原始指针。安全代码可以传递原始指针、比较它们，以及通过从引用（甚至整数）转换创建它们，但只有不安全代码才可以真正使用它们访问内存。21.7 节将详细介绍原始指针并解释如何安全地使用它们。
- 可以访问可修改 `static` 变量。正如 19.3.11 节所解释的，Rust 无法确定线程什么时候使用的是可修改 `static` 变量。因此它们的协议要求你保证所有访问必须适当同步过。
- 可以访问通过 Rust 的外来函数接口声明的函数和变量。即使是不可修改的，这些函数和变量也会被认为是不安全的。因为它们对其他语言写的代码是可见的，而那些代码不一定遵守 Rust 的安全规则。

将不安全特性限制在 `unsafe` 块并不会真正妨碍你做任何想做的事。你的代码完全可以只有一个 `unsafe` 块，然后想怎么写都行。这条规则的主要目的是引起人们注意，告诉大家 Rust 并不保证这里面代码的安全。

- 你不会意外使用不安全特性，然后发现自己要遵守根本都不知道的协议。
- 评审代码的人会格外关注 `unsafe` 块。某些项目甚至会自动把影响 `unsafe` 块的改动标示出来，以吸引特殊关注。

- 在写 unsafe 块时，你可以先思考片刻，问自己是不是真的需要采取这种手段。如果是为了性能，那有没有测试数据表明这实际上是一个瓶颈？也许使用安全 Rust 做同一件事才是最好的。

示例：高效ASCII字符串类型

下面展示了一个 Ascii 的定义，即一个字符串类型，确保其内容始终是有效的 ASCII。这个类型使用了一个不安全特性提供到 String 的零开销转换：

```
mod my_ascii {
    use std::ascii::AsciiExt;  // 为使用u8::is_ascii

    /// 一个ASCII编码字符串
    #[derive(Debug, Eq, PartialEq)]
    pub struct Ascii(
        // 这里必须只保存格式正确的ASCII文本：
        // 即字节从0到0x7f
        Vec<u8>
    );

    impl Ascii {
        /// 基于bytes中的ASCII文本创建Ascii。如果bytes中包含
        /// 任何非ASCII字符，则返回一个NotAsciiError错误
        pub fn from_bytes(bytes: Vec<u8>) -> Result<Ascii, NotAsciiError> {
            if bytes.iter().any(|&byte| !byte.is_ascii()) {
                return Err(NotAsciiError(bytes));
            }
            Ok(Ascii(bytes))
        }
    }

    // 转换失败时，返回这个不能转换的向量
    // 这里应该实现std::error::Error，为简单起见就省略了
    #[derive(Debug, Eq, PartialEq)]
    pub struct NotAsciiError(pub Vec<u8>);

    // 安全、高效的转换，使用不安全代码实现
    impl From<Ascii> for String {
        fn from(ascii: Ascii) -> String {
            // 如果模块没有bug，这是安全的，因为格式正确的
            // ASCII文本也是格式正确的UTF-8
            unsafe { String::from_utf8_unchecked(ascii.0) }
        }
    }
    ...
}
```

这个模块的关键是 Ascii 类型的定义。这个类型本身标记为 pub，因此对 my_ascii 模块外部是可见的。但它的 Vec<u8> 元素**不是**公有的，因此只有 my_ascii 模块可以构建 Ascii 值或引用其元素。这样，模块的代码就可以对出现或没出现在这里的代码拥有完全控制权。只要公有构造函数和方法保证新建的 Ascii 值格式正确，而且在它们的生命期内始终如此，

程序的其他部分就不会违反这个规则。而且确实，公有构造函数 `Ascii::from_bytes` 在根据传入的向量构建 `Ascii` 之前，也对其进行了仔细的检查。为了简洁起见，我们没有展示任何方法，但大家可以想象这里有一组文本处理方法，这些方法始终可以保证 `Ascii` 值包含正确的 ASCII 文本。就像 `String` 的方法保证它的内容始终是格式正确的 UTF-8 一样。

这样一来，就可以非常高效地为 `String` 实现 `From<Ascii>` 了。不安全函数 `String::from_utf8_unchecked` 接收一个字节向量，然后根据它构建一个 `String`，而无须检查其内容是不是格式正确的 UTF-8 文本。这个函数的协议要求调用者对此承担责任。所幸的是，`Ascii` 类型要求的规则也是我们需要满足 `from_utf8_unchecked` 协议的条件。正如 17.1.2 节所解释的，任何 ASCII 文本都是格式正确的 UTF-8，因此 `Ascii` 底层的 `Vec<u8>` 可以直接作为 `String` 的缓冲区使用。

有了以上定义，便可以写出下面的代码：

```
use my_ascii::Ascii;

let bytes: Vec<u8> = b"ASCII and ye shall receive".to_vec();

// 这个调用不需要分配内存或复制文本，只需一次扫描
let ascii: Ascii = Ascii::from_bytes(bytes)
    .unwrap(); // 我们知道这些字节都没问题

// 这个调用是零开销：无须分配、复制或扫描
let string = String::from(ascii);

assert_eq!(string, "ASCII and ye shall receive");
```

使用 `Ascii` 不要求 `unsafe` 块。我们已经使用不安全操作实现了一个安全接口，而且是否满足这个接口的协议只取决于模块自己的代码，与用户的行为无关。

`Ascii` 只是对 `Vec<u8>` 的包装，但隐藏了模块内部对其内容进行的额外检查。像这样的类型就叫作**新类型**（newtype），是 Rust 中一种常用模式。Rust 自己的 `String` 类型实际上也是这样定义的，区别在于其内容被限制为 UTF-8，而不是 ASCII。事实上，标准库中 `String` 的定义就是这样的：

```
pub struct String {
    vec: Vec<u8>,
}
```

在机器的层面并不区分 Rust 类型，新类型及其元素在内存中拥有相同的表示，因此构建新类型根本不需要任何机器指令。在 `Ascii::from_bytes` 中，表达式 `Ascii(bytes)` 对应的就是一个保存 `Ascii` 值的 `Vec<u8>`。类似地，在行内化之后，`String::from_utf8_unchecked` 可能不需要机器指令，因为此时 `Vec<u8>` 被认为是一个 `String`。

21.3 不安全的函数

`unsafe` 函数就是在普通函数定义前面加上 `unsafe` 关键字。`unsafe` 函数体自动被当成 `unsafe` 块。

只能在 `unsafe` 块中调用 `unsafe` 函数。这意味着将函数标记为 `unsafe` 会提醒调用者，使用

这个函数必须满足协议要求才能避免未定义行为。

例如，下面是为前面的 Ascii 类型定义的一个新构造函数，这个函数不管字节向量的内容是不是有效的 ASCII，就直接基于它构建 Ascii：

```
// 以下代码必须放到my_ascii模块中
impl Ascii {
    /// 基于bytes创建Ascii值，但不检查bytes中是否真正包含格式正确的ASCII
    ///
    /// 这个构造函数不做错误处理，直接返回一个Ascii，而不是像from_bytes
    /// 构造函数那样返回Result<Ascii, NotAsciiError>
    ///
    /// # 安全
    ///
    /// 调用者必须保证bytes只包含ASCII字符：即不大于0x7f的字节
    /// 否则，结果未定义
    pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
        Ascii(bytes)
    }
}
```

这个函数的假定是这样的：调用 Ascii::from_bytes_unchecked 的代码已经知道取得的向量中只包含 ASCII 字符，因此 Ascii::from_bytes 坚持要做的检查只是浪费时间，调用者还必须为此编写处理 Err 结果的代码，但这个结果永远不会发生。Ascii::from_bytes_unchecked 让这样的调用者可以避开有效性检查和错误处理。

但 Ascii 类型定义上面注释的意思是："这个模块不允许把非 ASCII 字节保存为 Ascii 值。"这个新的 from_bytes_unchecked 构造函数真的能够做到这一点吗？

并不确定。from_bytes_unchecked 的义务是根据自己的协议把这些字节传给调用者。正是这个协议才保证把它标记为 unsafe 是正确的，无论函数本身是否执行了不安全的操作，调用者都必须遵循 Rust 不会自动强制的规则以避免未定义行为。

真的可以违反 Ascii::from_bytes_unchecked 的协议而导致未定义行为吗？是的。可以像下面这样构建一个保存格式无效 UTF-8 的 String：

```
// 可以把这个向量想象成是执行某个复杂流程得到的结果
// 这个流程本来应该产生ASCII，但不知道哪里出问题了！
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];

let ascii = unsafe {
    // 因为bytes中包含非ASCII字节，所以违反了不安全
    // 函数的协议
    Ascii::from_bytes_unchecked(bytes)
};

let bogus: String = ascii.into();

// 现在bogus中保存着格式错误的UTF-8。解析其第一个字符
// 得到的char并不是一个有效的Unicode码点
assert_eq!(bogus.chars().next().unwrap() as u32, 0x1fffff);
```

这个例子反映出了有关 bug 和不安全代码的两个重要事实。

- **`unsafe` 块之前出现的 bug 可能破坏协议**。`unsafe` 块是否导致未定义行为并不只取决于这个块本身的代码，也取决于为它提供值的代码。`unsafe` 代码为满足协议而依赖的任何事物都关系到安全。基于 `String::from_utf8_unchecked` 实现的从 `Ascii` 到 `String` 的转换只有在模块其他部分正确维护 `Ascii` 不变性的前提下才是明确定义的。
- **破坏协议的后果可能在离开 `unsafe` 块后才会显现**。没有满足某个不安全特性的协议而导致的未定义行为通常不会在这个 `unsafe` 块本身发生。像前面构建的这个“假的”（bogus）字符串，可能要到程序真正执行以后很长时间才会导致问题。

本质上，Rust 类型检查器、借用检查器和其他静态检查机制在检查你的程序时，会尝试搜集证据，证明它不会出现未定义行为。Rust 能成功编译你的程序就意味着它找到了你的代码没问题的证据。但这个证明过程不包含 `unsafe` 块。因为 `unsafe` 块意味着你告诉 Rust：“这块代码没问题，相信我。”你的声明能否成真取决于程序中任何可能影响这个 `unsafe` 块的代码。而未能成真的后果可能会出现在被这个 `unsafe` 块影响的任何地方。写下 `unsafe` 关键字，就相当于提醒自己，你的程序不会全部享受到这个语言安全检查的好处。

即使有这个选择，也应该尽量创建安全的接口，不创造协议。这样会更简单，因为用户可以指望 Rust 的安全检查来保证自己的代码不会有未定义行为。就算你的实现使用了不安全特性，最好还是使用 Rust 的类型、生命期和模块系统来满足它们的协议，同时只使用你自己可以保证的部分，而不是把责任推给你的调用者。

可惜的是，未在文档中明确解释其协议的不安全函数并不鲜见。这就要求使用者凭借自己的经验和对代码行为的认知自行推断规则。如果你有过使用 C 或 C++ API 时那种内心不安的经历，那么应该就能体会这种心情。

21.4　不安全的块还是不安全的函数

有时候，你可能不知道自己到底该使用 `unsafe` 块还是该直接把函数都标记为 `unsafe`。建议你首先考虑函数。

- 如果会以编译通过但仍导致未定义行为的方式误用函数，则必须将其标记为不安全。正确使用函数的规则是它的协议，协议的存在是函数不安全的根本。
- 如果不会，那函数就是安全的。换句话说，只要类型正确的调用都不会导致未定义行为。这个函数就不应试标记为 `unsafe`。

函数是否安全与函数体内是否使用不安全特性无关。真正决定函数安全与否的是有无协议。前面已经展示了一个没有使用不安全特性的不安全函数，以及一个使用了不安全特性的安全函数。

不要只因为函数体内使用了不安全特性就把安全的函数标记为 `unsafe`。这样会导致函数更难使用，让人（自然）想去寻找在哪里可以看到有关其协议的解释。这时候要使用 `unsafe` 块，即使整个函数只有这一个块。

21.5 未定义行为

本章开头说过，**未定义行为**是指“Rust 坚决认为你的代码永远不会出现的行为”。这句话的措辞有点奇怪，特别是根据我们使用其他语言的经验，知道这些行为**确实会**以某种频率出现。为什么这个概念对界定不安全代码的责任有帮助？

编译器是一个将一种语言翻译为另一种语言的翻译器。Rust 编译器将 Rust 程序翻译为等价的机器语言程序。但是说两种以完全不同语言表示的程序等价，这意味着什么呢？

所幸的是，这个问题对程序员而言比对语言学家更容易理解。我们通常所说的两个程序等价，意味着它们在执行时会表现出相同的行为。比如会触发相同的系统调用、以相同的方式与外部库交互，等等。这有点像对程序进行图灵测试：如果你无法分辨自己是在跟原始版还是翻译版交流，那它们就是等价的。

现在来看看这段代码：

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

即使不知道 very_trustworthy 的定义，也可以看出它只接收一个对 i 的共享引用，因此这个调用不会改变 i 的值。因为传给 println! 的值始终都是 1000，所以 Rust 在把这段代码翻译成机器语言时，可以就当它是这么写的：

```
very_trustworthy(&10);
println!("{}", 1000);
```

这个变换之后的版本与原始版具有相同的可见行为，但可能稍微快一点。不过只有在保证它与原始版具有相同含义的前提下谈论性能才是有意义的。如果 very_trustworthy 的定义是下面这样呢？

```
fn very_trustworthy(shared: &i32) {
    unsafe {
        // 把共享引用转换为一个可修改指针
        // 这是一个未定义行为
        let mutable = shared as *const i32 as *mut i32;
        *mutable = 20;
    }
}
```

这段代码突破了共享引用的规则，它把 i 的值修改为 20，即使这个值应该被冻结，因为 i 是借来共享的。结果，我们在调用代码中所做的变换就有了不同的可见效果。如果 Rust 变换代码，程序就打印 1000；如果不做任何变换并使用 i 的新值，程序就打印 2000。在 very_trustworthy 中突破共享引用的规则意味着共享引用的行为在调用代码中不会再像预期那样了。

Rust 可能会执行的任何变换都有这个问题。即便把一个函数行内化到调用它的地方，也会假设在被调用代码完成时，控制流会返回调用的地方。（先不说其他假设。）但本章一开始就给出了一个违背这个假设，从而导致行为异常的例子。

对 Rust（或者任何其他语言）来说，要评估对程序的变换是否保留了其含义基本上是不可能的，除非它可以信任这门语言的基本特性具有预定义的行为。而到底这些特性是否具有预定义行为，不仅取决于手边的代码，还取决于程序中那些你看不到的部分。为了可以对你的代码做任何处理，Rust 必须假设你程序的其他部分都是行为正常的。

以下是 Rust 评判程序行为是否正常的标准。

- 程序必须不读取未初始化的内存。
- 程序必须不创建无效的原始值。
  - 引用或装箱的值是 `null`
  - `bool` 值不是 0 或 1
  - `enum` 值包含无效判别式（discriminant）值
  - `char` 值包含无效或代理对 Unicode 码点
  - `str` 值包含格式不正确的 UTF-8
- 程序必须遵守第 5 章解释的关于引用的规则。引用生命期不能超过其引用目标的生命期，共享访问是只读访问，可修改访问是专有（排他）访问。
- 程序必须不对空指针、未正确对齐的指针或悬空指针解引用。
- 程序必须不使用指针访问与该指针关联的分配内存区域之外的内存。21.7.1 节将详细解释这条规则。
- 程序必须没有数据争用。两个线程在访问没有同步的同一块内存且至少一个访问是写的情况下会发生数据争用。
- 程序必须不在另一种语言（通过外来函数接口执行）的调用期间展开，正如 7.1.1 节所解释的那样。
- 程序必须遵守标准库函数的协议。

这些规则都是 Rust 在优化你的程序并将其翻译成机器语言时假设成立的。简单来说，未定义行为就是指任何违反上述规则的行为。这也是我们会说 Rust 坚决认为你的代码永远不会出现未定义行为的原因。如果想要得出编译后的程序是对源代码忠实翻译的结论，这个假设就是必要的。

未使用不安全特性的 Rust 代码只要编译通过，就一定会遵守前面的所有规则。只有在使用不安全特性时，贯彻这些规则才会成为你的责任。在 C 和 C++ 中，你的程序可以没有任何错误或警告地编译通过说明不了什么。正如本书前面所提到的，即使那些因为广受赞誉的项目而青史留名的最优秀的 C 和 C++ 程序员，他们写的那些高水平的代码在实践中照样会出现未定义行为。

21.6 不安全的特型

不安全的特型指的是有 Rust 不能检查或强制的协议，实现者必须满足这些协议才能避免未定义行为的特型。要实现不安全特型，必须将实现标记为 `unsafe`。理解特型的协议并让你的类型满足该协议是你的责任。

通常，将其类型变量与不安全特型绑定的函数自身也会使用不安全特性，而满足它们的协议只能依赖于不安全特型的协议。对这个特型不正确的实现可能导致这个函数出现未定义行为。

不安全特型的经典例子是 `std::marker::Send` 和 `std::marker::Sync`。这两个特型没有定义任何方法，因此你可以用自己喜欢的任何类型轻松实现它们。但是它们有协议：`Send` 要求实现者可以安全地转移到另一个线程，而 `Sync` 要求实现者可以安全地通过共享引用在线程间共享。给不适当的类型实现 `Send`，可能会导致比如 `std::sync::Mutex` 不再安全，从而无法避免数据争用。

举一个例子，Rust 标准库中包含一个不安全特型 `core::nonzero::Zeroable`。这个特型用于让类型通过将它们的所有字节设置为 0 来实现安全初始化。很明显，`usize` 类型都是 0 没问题，但全为 0 的 `&T` 就是一个空引用，一旦解引用就会造成崩溃。对于可以用 0 初始化的类型，还可以应用一些优化，比如可以使用 `std::mem::write_bytes`（`memset` 在 Rust 中的等价特性）快速初始化这种类型的数组，或者使用分配 0 填充页的操作系统调用。（到 Rust 1.17 为止，`Zeroable` 还是实验性的，因此其可能会在未来的 Rust 版本中被修改或删除。不过它是一个简单、合适、真实的例子。）

`Zeroable` 是一个典型的标记特型（marker trait），没有方法和关联类型：

```
pub unsafe trait Zeroable {}
```

相应类型对它的实现也同样简单明了：

```
unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// 还有剩下的其他整数类型
```

有了这些定义，就可以写一个函数，让它快速分配一个指定长度的向量，包含 `Zeroable` 类型的值：

```
#![feature(nonzero)] // 许可Zeroable

extern crate core;
use core::nonzero::Zeroable;

fn zeroed_vector<T>(len: usize) -> Vec<T>
    where T: Zeroable
{
    let mut vec = Vec::with_capacity(len);
    unsafe {
        std::ptr::write_bytes(vec.as_mut_ptr(), 0, len);
        vec.set_len(len);
    }
    vec
}
```

这个函数先创建了一个包含所需容量的空 `Vec`，然后调用 `write_bytes` 将未占用的缓冲区填上 0。（`write_bytes` 函数将 `len` 理解为 `T` 元素的数量，而非字节数，因此这个调用会填满整个缓冲区。）向量的 `set_len` 方法只会修改其长度，不会对缓冲区做任何事情。这是不安全的，因为必须保证新的缓冲区实际包含类型 `T` 适当初始化的值。不过这正是绑定 `T: Zeroable` 所保证的：以 0 填充的一堆字节表示一个有效的 `T` 值。我们对 `set_len` 的使用是安全的。

接下来使用这个函数：

```
let v: Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));
```

显然，Zeroable 一定是不安全特型，因为不遵守其协议的实现可能导致未定义行为：

```
struct HoldsRef<'a>(&'a mut i32);

unsafe impl<'a> Zeroable for HoldsRef<'a> { }

let mut v: Vec<HoldsRef> = zeroed_vector(1);
*v[0].0 = 1;   // 崩溃：解引用空指针
```

Rust 会毫无怨言地编译这段代码，它对 Zeroable 要表示什么没有想法，因此也不知道它什么时候会在不适当的类型上实现。与其他不安全特性一样，理解并遵守不安全特型的协议是你的责任。

注意，不安全代码不能依赖普通、安全特型的正确实现。假设有一个 std::hash::Hasher 特型的实现，它简单地返回一个随机散列值，与用来计算散列的值无关。这个特型要求对同一个位进行两次散列运算必须产生相同的散列值。但这个实现不满足这个要求，这绝对是错误的。但因为 Hasher 并非不安全特型，不安全代码在使用这个散列器时必须不能出现未定义行为。std::collections::HashMap 类型认真遵守了它所使用的不安全特性的协议，而没有考虑散列器的行为。可以肯定，这个散列表不能正确运行：查询会失败，条目也会随机出现和消失。但这个表不会出现未定义行为。

21.7　原始指针

Rust 中的**原始指针**是一种不受约束的指针。使用原始指针可以创建 Rust 检查的指针类型所不能创建的任何结构，比如双向链表或任意对象的图。但由于原始指针过于灵活，Rust 无法判断你是否在安全地使用它们，因此只能在 unsafe 块中对它们解引用。

原始指针本质上等价于 C 或 C++ 指针，因此也常用于操作这些语言编写的代码。

有两种原始指针：

- *mut T 是一个指向 T 且允许修改其引用目标的原始指针；
- *const T 是一个指向 T 但只允许读取其引用目标的原始指针。

（没有简单的 * T 类型，必须始终指定 const 或 mut。）

可以把引用转换为原始指针，然后使用 * 操作符对其解引用：

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;

let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
    *ptr_x += *ptr_y;
```

```
}
assert_eq!(x, 30);
```

与装箱和引用不同，原始指针可以是空的，类似 C 中的 NULL 或 C++ 中的 nullptr：

```
fn option_to_raw<T>(opt: Option<&T>) -> *const T {
    match opt {
        None => std::ptr::null(),
        Some(r) => r as *const T
    }
}

assert!(!option_to_raw(Some(&("pea", "pod"))).is_null());
assert_eq!(option_to_raw::<i32>(None), std::ptr::null());
```

这个例子中没有 unsafe 块，因为创建原始指针、传递原始指针，以及比较原始指针都是安全的。只有解引用原始指针是不安全的。

指向非固定大小类型的原始指针是一个胖指针，就像对应的引用或 Box 类型一样。*const [u8] 指针包含长度和地址，类似 *mut std::io::Write 这样特型对象的指针则带有一个虚拟表。

Rust 虽然会在各种情况下隐式解引用安全指针类型，但原始指针必须显式解引用。

- 操作符 . 不会隐式解引用原始指针，必须写成 (*raw).field 或 (*raw).method(...)。
- 原始指针没有实现 Deref，因此 Deref 转换不适合它们。
- 操作符 == 和 < 等比较原始指针的地址，即如果两个原始指针指向内存中的相同位置那它们就相等。类似地，对原始指针计算散列值，是对其指向的地址而非引用目标计算散列值。
- 格式化特型 std::fmt::Display 等会自动跟随引用，但它们根本不处理原始指针。std::fmt::Debug 和 std::fmt::Pointer 例外，它们显示原始指针的十六进制地址，但不会对它们解引用。

与 C 和 C++ 中的 + 操作符不同，Rust 的 + 操作符不处理原始指针。不过，你可以通过原始指针的 offset 和 wrapping_offset 方法实现指针算术。没有像 C 和 C++ 中的 - 操作符那样的返回两个指针距离的标准方法，不过，你可以自己写一个：

```
fn distance<T>(left: *const T, right: *const T) -> isize {
    (left as isize - right as isize) / std::mem::size_of::<T>() as isize
}

let trucks = vec!["garbage truck", "dump truck", "moonstruck"];
let first = &trucks[0];
let last = &trucks[2];
assert_eq!(distance(last, first), 2);
assert_eq!(distance(first, last), -2);
```

即使 distance 的参数是原始指针，我们也可以把引用传给它。Rust 会隐式地将引用转换为原始指针（当然，不会反向转换）。

Rust 的 as 操作符可以将引用转换为原始指针，或者将一种原始指针转换为另一种原始指

针，只要看起来有可能，几乎都没问题。不过，可能需要将一个复杂的转换分成几个简单的步骤。比如：

```
&vec![42_u8] as *const String // 错误：无效转换
&vec![42_u8] as *const Vec<u8> as *const String; // 可以转换
```

注意，as 操作符不能把原始指针转换为引用。这种转换不安全，as 只能用于安全操作。为此，必须先（在 unsafe 块中）对原始指针解引用，然后再借用得到的值。

这样操作时一定要小心。以这种方式产生的引用具有不受限的生命期，也就是说它能“活”多长时间没有限制，因为原始指针不会给 Rust 做这种决定的参照。本章后面的 21.8.5 节将展示几个适当限制生命期的例子。

很多类型有 as_ptr 和 as_mut_ptr 方法，它们返回指向其内容的原始指针。例如，数组切片和字符串返回的指针指向它们的第一个元素，而有些迭代器返回的指针会指向它们产生的下一个元素。像 Box、Rc 和 Arc 这样的所有型指针类型有 into_raw 和 from_raw 函数，可以实现与原始指针之间的相互转换，其中某些方法的协议会有一些出人意料的要求，因此在使用之前最好先查看一下它们的文档。

我们也可以把整数转换为原始指针，虽然唯一可以信任的整数是那些原本就源自指针的整数。21.7.2 节将展示一个这样使用原始指针的例子。

与引用不同，原始指针既不是 Send 也不是 Sync。因此，任何包含原始指针的类型默认都不会实现这两个特型。跨线程发送或共享原始指针不存在固有的不安全性。毕竟无论它们跑到哪里，你都需要在一个 unsafe 块中解引用它们。但考虑到原始指针通常所扮演的角色，这门语言的设计者认为这种行为最好是默认的。前面 21.6 节已经讨论过如何实现 Send 和 Sync 了。

21.7.1 安全解引用原始指针

以下是安全使用原始指针的一些常识性提示。

- 解引用空指针或悬空指针是未定义行为，与引用未初始化内存或离开作用域的值一样。
- 解引用没有与它们引用目标类型正确对齐的指针是未定义行为。
- 如果想从解引用后的原始指针中借用值，必须遵守第 5 章讲解的引用安全规则：引用不应该“活”得比它们的引用目标还长，共享引用是只读访问，可修改引用是专有访问。（这个规则很容易意外违反，因为原始指针经常用于创建包含非标准共享或所有权的数据结构。）
- 如果想使用原始指针的引用目标，必须保证它是该类型格式正确的值。例如，必须确保解引用 *const char 以后得到的是一个正确的、非代理对 Unicode 码点。
- 如果想在原始指针上使用 offset 和 wrapping_offset 方法，必须保证指针只指向变量或原来指针所指向的堆内存块中的字节，或者指向这个范围后面的第一个字节。

  如果要做指针算术，把原始指针转换为整数，用整数来做计算，再把计算结果转换为指针，那么结果指针必须是 offset 方法的规则允许产生的。
- 如果要给原始指针的引用目标赋值，则必须不能违反引用目标所在类型的不变性。例如，要是你有一个 *mut u8 指向某个 String 的一个字节，那么在这个 u8 中存储的值必须保证该 String 持有格式正确的 UTF-8。

除了借用规则，这些本质上都是在使用 C 或 C++ 指针时必须遵循的规则。

不违反类型不变性的原因应该说清楚。很多 Rust 标准类型在自己的实现中使用不安全代码，但仍然提供安全接口，假定它们可以通过 Rust 的安全检查，并遵守其模块系统和可见性规则。使用原始指针绕过这些保护措施可能导致未定义行为。

原始指针完整、准确的协议并不容易说清楚，而且还会随着语言的发展而改变。但这里概述的原理对保证代码的安全性还是适用的。

21.7.2　示例：RefWithFlag

下面这个例子使用原始指针实现了经典的[1]位级黑科技，并将其包装为一个彻底安全的 Rust 类型。这个模块定义了一个类型，即 `RefWithFlag<'a, T>`，它保存着一个 `&'a T` 和一个 `bool`，类似于元组 `(&'a T, bool)`，但仍然只设法占用一个机器字，而不是两个。这种技术经常用于垃圾收集器和虚拟机的实现，其中某些类型（比如表示对象的类型）因为实在太多，结果就算只给每个值增加一个字都会明显增加内存占用。

```
mod ref_with_flag {
    use std::marker::PhantomData;
    use std::mem::align_of;

    /// 包装在一个字中的&T和bool
    /// 类型T必须要求至少二字节对齐
    ///
    /// 作为程序员，如果你还从未遇到过一个不想偷走的2°位的指针，
    /// 那你现在可以安全地这样做了！（“不过这样就没那么刺激了……”）
    pub struct RefWithFlag<'a, T: 'a> {
        ptr_and_bit: usize,
        behaves_like: PhantomData<&'a T> // 不占空间
    }

    impl<'a, T: 'a> RefWithFlag<'a, T> {
        pub fn new(ptr: &'a T, flag: bool) -> RefWithFlag<T> {
            assert!(align_of::<T>() % 2 == 0);
            RefWithFlag {
                ptr_and_bit: ptr as *const T as usize | flag as usize,
                behaves_like: PhantomData
            }
        }

        pub fn get_ref(&self) -> &'a T {
            unsafe {
                let ptr = (self.ptr_and_bit & !1) as *const T;
                &*ptr
            }
        }

        pub fn get_flag(&self) -> bool {
            self.ptr_and_bit & 1 != 0
```

注 1：好吧，应该说是成就我们的经典。

```
            }
        }
    }
```

以上代码利用了很多类型在内存中必须放在偶数地址的事实。因为偶数地址的最低有效位始终为 0，所以可以在那里存点别的，然后只要对最低位应用掩码就能可靠地重建原始地址。但并不是所有类型都可以，比如类型 u8 和 (bool, [i8; 2]) 可以放在任意地址。但我们可以在构造时检查类型的对齐，拒绝不合适的类型。

可以像这样使用 RefWithFlag：

```
use ref_with_flag::RefWithFlag;

let vec = vec![10, 20, 30];
let flagged = RefWithFlag::new(&vec, true);
assert_eq!(flagged.get_ref()[1], 20);
assert_eq!(flagged.get_flag(), true);
```

构造函数 RefWithFlag::new 接收一个引用和一个 bool 值，保证该引用的类型是合适的，然后把引用转换为一个原始指针，再转换为一个 usize。usize 类型的大小足以容纳作为编译目标的任何处理器下的指针，因此把原始指针转换为 usize 再转换回来是定义明确的。得到 usize 之后，我们知道它一定是偶数，因此可以使用按位或操作符 | 将它与 bool 组合，而这个 bool 已经被转换成了整数 0 或 1。

get_flag 方法用于提取 RefWithFlag 的 bool 组件。很简单，只要对最低位应用掩码并检查它是否非零即可。

get_ref 方法用于从 RefWithFlag 中提取引用。首先，对这个 usize 的最低位应用掩码并将其转换为一个原始指针。as 操作符不会将原始指针转换为引用，但我们可以解引用这个原始指针（当然是在 unsafe 块中），然后再借用它。借用原始指针的引用目标会得到一个生命期无限长的引用。Rust 会赋予引用一个满足其周围代码要求的生命期（如果有的话）。不过，通常会有一些比较明确的生命期，这些生命期过于精确，因而会碰到更多错误。在这个例子中，get_ref 的返回类型是 &'a T。Rust 推断这个引用的生命期一定是 RefwithFlag::new 的参数，这正是我们想要的，因为这个生命期就是原本引用的生命期。

在内存中,RefWithFlag 看起来就是一个 usize。因为 PhantomData 是一个没有大小的（zero-sized）类型，behaves_like 字段在这个结构体中不占空间。但这个 PhantomData 是必要的，因为在使用 RefWithFlag 的代码中，Rust 要通过它才能知道如何处理生命期。想象一下这个类型没有 behaves_like 字段的样子：

```
// 这样不会编译
pub struct RefWithFlag<'a, T: 'a> {
    ptr_and_bit: usize
}
```

第 5 章曾指出过，任何包含引用的结构体必须不能“活”得比它们借用的值还要长，以免引用变成悬空指针。这个结构体必须遵守应用到其字段的限制。这个限制当然也会应用到 RefWithFlag。在刚刚看到的示例代码中，flagged 必须不能比 vec“长寿”，因为 flagged.get_ref() 返回一个对它的引用。但这里精简后的 RefWithFlag 类型根本不包含

引用，永远不会使用它的生命期参数 `'a`，它就是一个 `usize`。Rust 如何才能知道 `ptr_and_bit` 有合适的生命期呢？包含一个 `PhantomData<&'a T>` 字段就是为了告诉 Rust，让它**视同** `RefWithFlag<'a, T>` 就包含一个 `&'a T`。而且这样实际也不影响这个结构体的表现。

尽管 Rust 并不真的知道发生了什么（这也是 `RefWithFlag` 不安全的原因），但它会尽力帮你处理生命期。如果省略这个 `behaves_like` 字段，Rust 则会抱怨参数中的 `'a` 和 `T` 没有被使用，并建议使用一个 `PhantomData`。

`RefWithFlag` 使用了与前面展示的 `Ascii` 类型一样的策略来避免其 `unsafe` 块的未定义行为。这个类型本身是 `pub` 的，但其字段不是，意味着只有 `ref_with_flag` 模块中的代码可以创建或查看 `RefWithFlag` 的值。不用看太多代码就可以确定 `ptr_and_bit` 字段是结构严谨的。

21.7.3　可空指针

Rust 中的空原始指针是一个零地址，与 C 和 C++ 中一样。对任意类型 `T`，`std::ptr::null<T>` 函数返回一个 `*const T` 空指针，而 `std::ptr::null_mut<T>` 返回一个 `*mut T` 空指针。

检查一个原始指针是否为空有几种方法。最简单的是使用 `is_null` 方法，但使用 `as_ref` 方法可能更方便。后者以一个 `*const T` 指针为参数，返回一个 `Option<&'a T>`，空指针会被转换为 `None`。类似地，`as_mut` 方法把 `*mut T` 指针转换为 `Option<&'a mut T>` 值。

21.7.4　类型大小与对齐

任何 `Sized` 类型的值在内存中都会占用固定数量的字节，而且必须放到一个是某个**对齐**（alignment）值倍数的地址上，对齐值由机器架构决定。例如，`(i32, i32)` 元组占用 8 字节，而大多数处理器倾向于将它放在一个是 4 的倍数的地址。

调用 `std::mem::size_of::<T>()` 返回类型 `T` 值的大小，以字节为单位。而调用 `std::mem::align_of::<T>()` 返回类型 `T` 要求的对齐值。例如：

```
assert_eq!(std::mem::size_of::<i64>(), 8);
assert_eq!(std::mem::align_of::<(i32, i32)>(), 4);
```

任何类型的对齐值都是 2 的幂。

即使技术上可以占用更少的空间，类型的大小也总是会四舍五入到其对齐值的倍数。例如，尽管元组 `(f32, u8)` 只占用 5 字节，但 `size_of::<(f32, u8)>()` 依然返回 8，因为 `align_of::<(f32, u8)>()` 是 4。这样可以确保如果你有一个数组，其元素类型的大小始终对应于两个元素的距离。

对于无固定大小的类型，其大小和对齐取决于当前值。对于一个无固定大小值的引用，`std::mem::size_of_val` 和 `std::mem::align_of_val` 函数返回这个值的大小和对齐值。这两个函数可用于 `Sized` 和非固定大小类型的引用。

```
// 指向切片的胖指针包含其引用目标的长度
let slice: &[i32] = &[1, 3, 9, 27, 81];
assert_eq!(std::mem::size_of_val(slice), 20);
```

```
let text: &str = "alligator";
assert_eq!(std::mem::size_of_val(text), 9);

use std::fmt::Display;
let unremarkable: &Display = &193_u8;
let remarkable: &Display = &0.0072973525664;

// 下面返回特型对象指向值（而非特型对象）的大小/对齐
// 这些信息来自特型对象引用的虚拟表
assert_eq!(std::mem::size_of_val(unremarkable), 1);
assert_eq!(std::mem::align_of_val(remarkable), 8);
```

21.7.5　指针算术

在内存中，Rust 把数组、切片或向量的元素排列为一个连续的块，如图 21-1 所示。元素是均匀分布的，因此每个元素占用 `size` 字节，而第 `i` 个元素开始于第 `i * size` 字节。

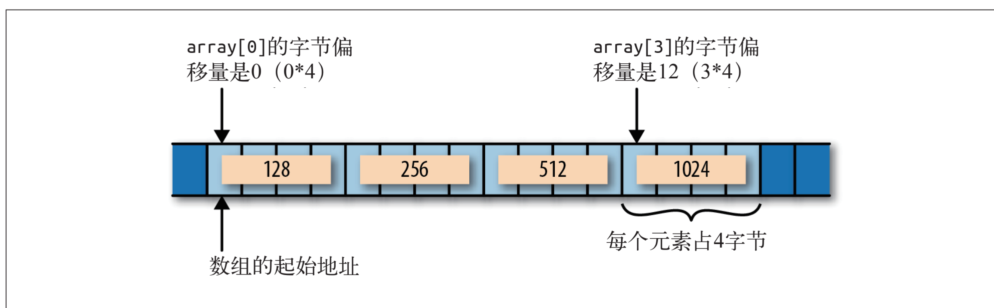


图 21-1：内存中的数组

这样的一个好处是，如果你有两个指向数组元素的原始指针，那么比较指针可以得到与比较元素索引一样的结果。如果 `i < j`，那么指向第 `i` 个元素的原始指针也小于指向第 `j` 个元素的原始指针。这样在遍历数组时，就可以使用原始指针作为界限。事实上，标准库中对切片的简单迭代器就是这样定义的：

```
struct Iter<'a, T: 'a> {
    ptr: *const T,
    end: *const T,
    ...
}
```

`ptr` 字段指向迭代应该产生的下一个元素，而 `end` 字段是一个界限：当 `ptr == end` 时，迭代完成。

数组布局的另一个好处是，如果 `element_ptr` 是指向某个数组第 `i` 个元素的 `*const T` 或 `*mut T` 原始指针，那么 `element_ptr.offset(o)` 就是指向第 `(i + o)` 个元素的原始指针，其定义等价于如下代码：

```
fn offset(self: *const T, count: isize) -> *const T
    where T: Sized
{
```

```
        let bytes_per_element = std::mem::size_of::<T>() as isize;
        let byte_offset = count * bytes_per_element;
        (self as isize).checked_add(byte_offset).unwrap() as *const T
    }
```

std::mem::size_of::<T> 函数返回类型 T 的字节大小。因为按照定义 isize 足够保存一个地址，所以可以将一个基础指针转换为一个 isize，对得到的值执行算术运算，然后再把结果转换为一个指针。

可以产生指向数组末尾之后第一个字节的指针。虽然不能解引用这样的指针，但可以用它来表示循环的上限，或者用于边界检查。

不过，使用 offset 产生一个指向这个位置之后或者指向数组开头之前的指针是未定义行为，即使你从来没有解引用它。为方便优化，Rust 会假设在 i 为正值是 ptr.offset(i) > ptr，在 i 为负值是 ptr.offset(i) < ptr。这个假设看起来安全，但在 offset 计算 isize 值溢出时可能就不成立了。如果将 i 限制在与 ptr 相同的数组中，则不会发生溢出，毕竟数组本身不会溢出地址空间的边界。（为了让指向末尾之后第一个字节的指针安全，Rust 永远不会在地址空间的上端保存值。）

如果确实需要将指针偏移出与之关联数组的这个界限，可以使用 wrapping_offset 方法。这个方法与 offset 等价，但 Rust 不会假设 ptr.wrapping_offset(i) 和 ptr 本身的相对顺序。当然，除非偏移后的指针落在数组范围内，否则还是不能解引用这种指针。

21.7.6 移入和移出内存

如果你正在实现的类型需要管理自己的内存，那么就要跟踪内存的哪一部分保存着活的值，哪个是未初始化的，就像 Rust 对局部变量所做的那样。来看如下代码：

```
    let pot = "pasta".to_string();
    let plate;

    plate = pot;
```

上面代码运行之后的内存状态如图 21-2 所示。

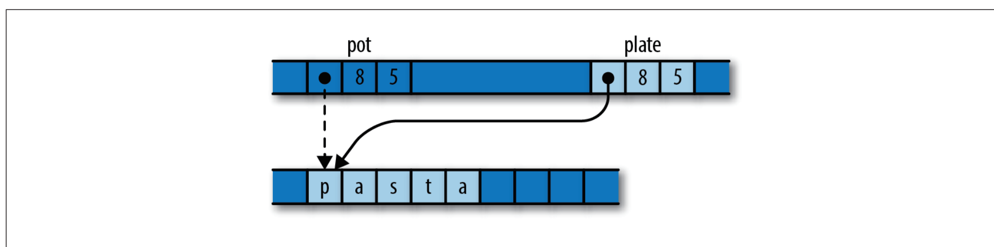


图 21-2：将字符串从一个局部变量转移到另一个局部变量

赋值之后，pot 变成了未初始化，而 plate 拥有了这个字符串。

在机器层面上，转移要对源做什么并没有规定，而实践中通常什么也不做。以上赋值可能仍然会让 pot 持有对字符串的指针、容量和长度。自然，如果再把它也看成活值将是灾难

性的，而 Rust 可以保证你不会这样做。

同样的情形也适用于管理自己内存的数据结构。假设运行了如下代码：

```
let mut noodles = vec!["udon".to_string()];
let soba = "soba".to_string();
let last;
```

内存中的状态如图 21-3 所示。

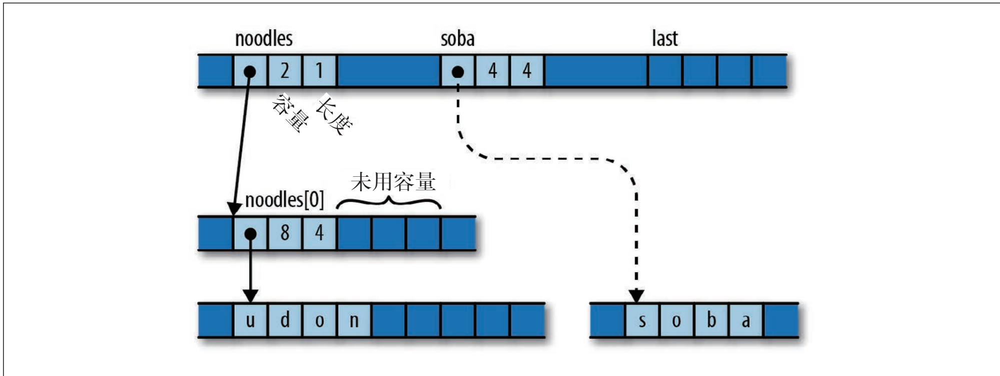


图 21-3：具有未初始化、空闲容量的向量

这个向量有空闲容量保存另一个元素，但其内容是“垃圾”，可能是该内存之前保存的东西。假设又运行了如下代码：

```
noodles.push(soba);
```

把新字符串推到向量中将未初始化内存转换为了新元素，如图 21-4 所示。

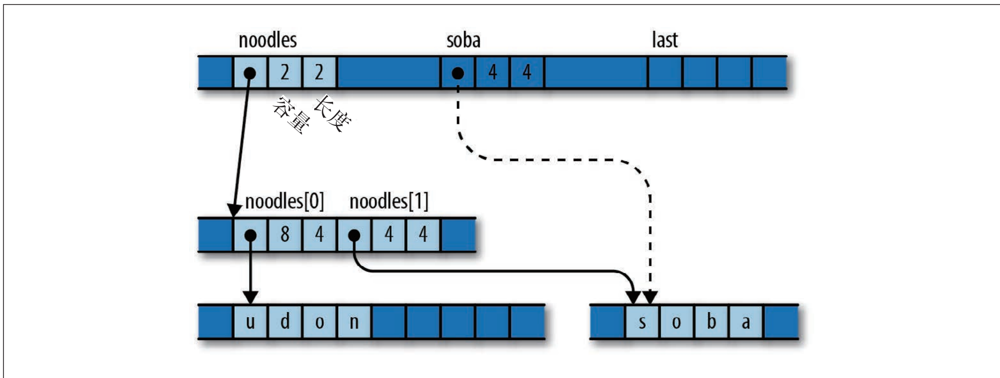


图 21-4：把 soba 值推到向量中之后

向量初始化了空闲容量以拥有这个字符串，并将其长度加 1 表明增加了一个新的活元素。这个向量成了这个字符串的所有者，现在可以引用其第二个元素了，而清除向量也会释放两个字符串。但 soba 现在变成了未初始化状态。

最后，再看看从向量中弹出一个值会怎么样：

```
last = noodles.pop().unwrap();
```

在内存中，现在的状态如图 21-5 所示。

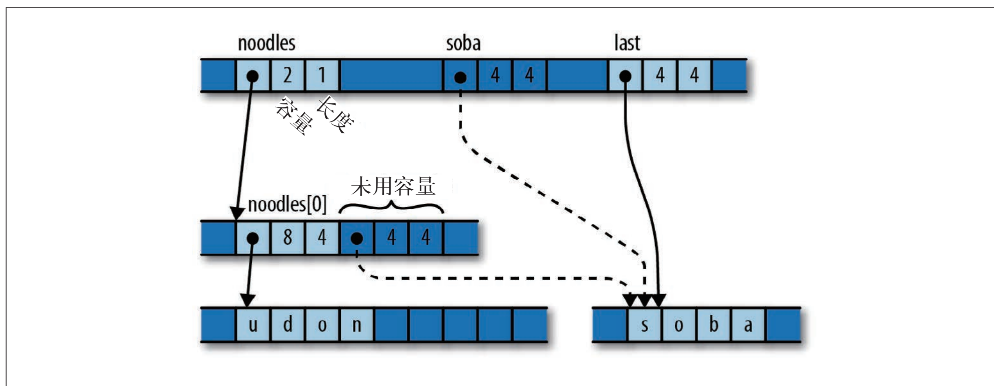


图 21-5：把向量中的元素弹出到 `last` 之后

变量 `last` 现在取得了这个字符串的所有权。向量将容量减 1 表示原来用于保存字符串的空间现在未初始化。

与前面的 `pot` 和 `pasta` 一样，`soba`、`last` 和向量空闲空间的位模式可能是相同的。但只有 `last` 才拥有这个值。把另外两个中的任何一个当成活值都不正确。

初始化值真正的定义是将其**作为活值**（treated as live）。写入值的字节通常是初始化的一部分，但这一部分仅仅是为了将其作为活值做的准备。

Rust 在编译时会跟踪局部变量。`Vec`、`HashMap`、`Box` 等类型会动态跟踪自己的缓冲区。如果你也要实现一个管理自己内存的类型，那需要做相同的事。

Rust 提供了两个基本操作以便实现这样的类型。

- **`std::ptr::read(src)`** 移出 `src` 指向位置的值，将所有权转移给调用者。调用 `read` 之后，必须将 `*src` 作为未初始化内存。`src` 参数应该是一个 `*const T` 原始指针，其中 `T` 是一个固定大小类型。

  `Vec::pop` 在后台就是执行的这个操作。弹出值会调用 `read` 将值移出缓冲区，然后减少长度以便将该空间标记为未初始化容量。

- **`std::ptr::write(dest, value)`** 把 `value` 移入 `dest` 指向的位置，该位置在这个调用前必须是未初始化内存。引用目标随后拥有了这个值。这里的 `dest` 必须是一个 `*mut T` 原始指针，而 `value` 必须是一个 `T` 值，其中 `T` 是固定大小类型。

  `Vec::push` 在后台就是执行的这个操作。推入值会调用 `write` 将值移入下一个可用空间，然后增加长度以便将该空间标记为有效元素。

这两个都是自由函数，不是原始指针类型上的方法。

注意，不能使用 Rust 的安全指针类型完成上述操作。安全指针类型要求其引用目标必须始终是初始化的，因此转换未初始化内存为值或相反的操作超出了这个范围。原始指针可以满足这个需求。

标准库也提供了将值的数组从一个内存块转移到另一个内存块的函数。

- **std::ptr::copy(src, dst, count)** 将 count 个值的数组在内存中从 src 转移到 dst，就像手写循环每次 read 和 write 一个元素一样。调用前目标内存必须是未初始化的，调用后源内存变量未初始化。src 和 dst 参数必须是 *const T 和 *mut T 原始指针，count 必须是 usize。
- **std::ptr::copy_nonoverlapping(src, dst, count)** 与相应的 copy 调用类似，只不过它的协议进一步要求源和目标内存块必须不能重叠。这可能比调用 copy 稍微快一点。

下面是另外两组 read 和 write 函数，它们也在 std::ptr 模块中。

- **read_unaligned** 和 **write_unaligned** 函数类似于 read 和 write，只不过指针不需要像要求常规引用目标那样对齐。这两个函数可能比纯 read 和 write 函数慢一点。
- **read_volatile** 和 **write_volatile** 函数是 C 或 C++ 中 volatile 读和写的对应函数。

21.7.7　示例：GapBuffer

下面这个例子展示了如何使用前面介绍的原始指针函数。

假设你在编写一个文本编辑器，正在寻找表示文本的类型。可以选择 String，使用 insert 和 remove 方法在用户输入时插入和删除字符。但如果是在一个大文件开头编辑文本，那这些方法开销会比较大：插入新字符涉及在内存中将所有其他字符串都向右移动，删除则会反向向左移动。我们希望这些常用操作开销小一些。

Emacs 编辑器使用了一个名为**间隙缓冲**（gap buffer）的简单数据结构，该数据结构插入和删除字符的时间为常量值。相比于 String 将其所有空闲容量放在文本末尾，使得 push 和 pop 操作很快，间隙缓冲则会将其空闲容量放在文本中间，编辑都在中间发生。这样的空闲容量被称为**间隙**。在间隙插入和删除元素开销很小，按需要简单地收缩或放大间隙即可。通过将文本从间隙一侧移动到另一侧可以把间隙移动到任意位置。在间隙为空时，再迁移到一个更大的缓冲区。

虽然在间隙缓冲中插入和删除很快，但改变操作的位置需要将间隙移动到新位置。移动元素所需时间与移动距离成正比。好在，典型的编辑活动都是在这个缓冲区附近做一通修改，在其他地方修改的概率不高。

本节将用 Rust 实现一个间隙缓冲。为避免因 UTF-8 而分心，我们直接让缓冲区存储 char 值。但操作的原理与使用其他形式存储文本相同。

首先，我们会展示一下间隙缓冲的实际应用。以下代码创建了一个 GapBuffer，向其中插入一些文本，然后把插入点移动到恰好位于最后一个词之前：

```
use gap::GapBuffer;

let mut buf = GapBuffer::new();
buf.insert_iter("Lord of the Rings".chars());
buf.set_position(12);
```

运行以上代码后，缓冲区类似图 21-6 所示。

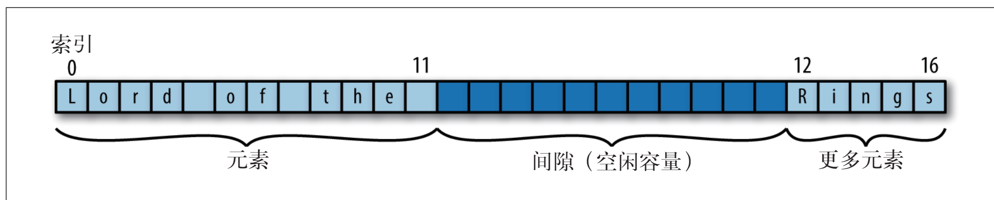


图 21-6：包含一些文本的间隙缓冲

插入涉及用新文本填充间隙。以下代码添加了一个词并破坏了这个“胶卷”：

```
buf.insert_iter("Onion ".chars());
```

结果内存状态如图 21-7 所示。

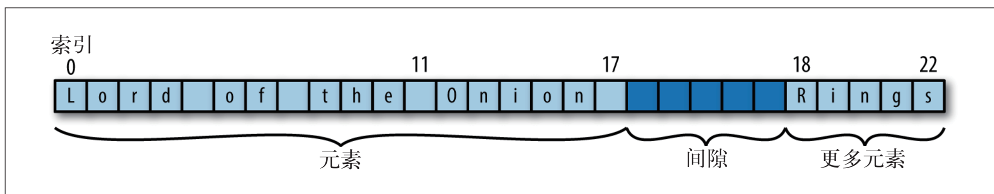


图 21-7：包含更多文本的间隙缓冲

以下是 GapBuffer 类型：

```
mod gap {
    use std;
    use std::ops::Range;

    pub struct GapBuffer<T> {
        // 元素存储。这个存储有我们需要的容量，但它的长度始终为0
        // GapBuffer将其元素和间隙放到了这个Vec的“未用”容量中
        storage: Vec<T>,

        // 未初始化元素的范围在storage的中间
        // 在这个范围之前和之后的元素始终是初始化的
        gap: Range<usize>
    }

    ...
}
```

GapBuffer 以一种奇怪的方式使用其 storage 字段。[2] 它不会真正在这个向量中存储任何元素（或者说基本不存储），而只是调用 Vec::with_capacity(n) 取得一块足够放下 *n* 个值的内存，通过这个向量的 as_ptr 和 as_mut_ptr 方法获得指向该内存的原始指针，然后按照自己的逻辑使用这个缓冲区。这个向量的长度始终为 0。当这个 Vec 被清除时，它会释放对应的内存块，而不会释放自己的元素，因为它不知道自己有元素。这正是 GapBuffer 想要的，它有自己的 Drop 实现，知道活的元素都放在哪里，因此可以正确清除它们。

GapBuffer 最简单的方法也是我们想要的：

```
impl<T> GapBuffer<T> {
    pub fn new() -> GapBuffer<T> {
        GapBuffer { storage: Vec::new(), gap: 0..0 }
    }

    /// 返回在不重新分配的前提下这个GapBuffer可以容纳的元素数量
    pub fn capacity(&self) -> usize {
        self.storage.capacity()
    }

    /// 返回这个GapBuffer当前容纳的元素数量
    pub fn len(&self) -> usize {
        self.capacity() - self.gap.len()
    }

    /// 返回当前插入位置
    pub fn position(&self) -> usize {
        self.gap.start
    }

    ...
}
```

它还在很多下面这样的函数中，通过一个实用方法返回给定索引对应的缓冲区元素的原始指针。这就是 Rust，最终我们需要一个返回 mut 指针的方法和一个返回 const 指针的方法。与前面的方法不同，这些方法不是公有的。继续前面的 impl 块：

```
/// 返回底层存储中第index个元素的指针，与间隙无关
///
/// 安全：index必须是self.storage中的一个有效索引
unsafe fn space(&self, index: usize) -> *const T {
    self.storage.as_ptr().offset(index as isize)
}
/// 返回底层存储中第index个元素的可修改指针，与间隙无关
///
/// 安全：index必须是self.storage中的一个有效索引
unsafe fn space_mut(&mut self, index: usize) -> *mut T {
    self.storage.as_mut_ptr().offset(index as isize)
}
```

要找到给定索引的元素，必须考虑这个索引在间隙之前还是之后，然后再适当调整：

注 2：处理这个的更好方式是使用 alloc 包的 RawVec 类型，但这个包在写作本书时还不稳定。

```
/// 返回第index个元素在缓冲区中的偏移量，考虑间隙的存在
/// 这个方法不检查索引是否在范围内，但永远不会返回间隙中的索引
fn index_to_raw(&self, index: usize) -> usize {
    if index < self.gap.start {
        index
    } else {
        index + self.gap.len()
    }
}

/// 返回对第index个元素的引用，如果index越界则返回None
pub fn get(&self, index: usize) -> Option<&T> {
    let raw = self.index_to_raw(index);
    if raw < self.capacity() {
        unsafe {
            // 只检查raw与self.capacity()，index_to_raw会跳过间隙，
            // 所以这样是安全的
            Some(&*self.space(raw))
        }
    } else {
        None
    }
}
```

当在缓冲区的不同部分执行插入和删除操作时，需要把间隙移动到新位置。向右移动间隙意味着把元素向左移动，反之亦然。这就像水平尺中的气泡，当液体朝一个方向流动时，气泡会朝另一个方向移动：

```
/// 将当前插入位置设置为pos
/// 如果pos越界，则诧异
pub fn set_position(&mut self, pos: usize) {
    if pos > self.len() {
        panic!("index {} out of range for GapBuffer", pos);
    }

    unsafe {
        let gap = self.gap.clone();
        if pos > gap.start {
            // pos落在间隙后面。通过将间隙后面的元素移动到间隙的
            // 前面来让间隙向右移动
            let distance = pos - gap.start;
            std::ptr::copy(self.space(gap.end),
                           self.space_mut(gap.start),
                           distance);
        } else if pos < gap.start {
            // pos落在间隙前面。通过将间隙前面的元素移动到间隙的
            // 后面来让间隙向左移动
            let distance = gap.start - pos;
            std::ptr::copy(self.space(pos),
                           self.space_mut(gap.end - distance),
                           distance);
        }

        self.gap = pos .. pos + gap.len();
    }
}
```

这个函数使用 std::ptr::copy 方法移动元素。copy 要求目标内存未初始化，最后让源内存变成未初始化。源和目标范围可能重叠，但 copy 可以正确处理这种情况。因为间隙在这个调用前是未初始化的内存，所以这个函数会调整间隙的位置以覆盖被这次复制腾出的空间，从而满足 copy 函数的协议。

元素的插入和删除相对简单。插入接收间隙的一块空间用于容纳新元素，删除则将值移出并扩大间隙以覆盖其原来占用的空间：

```
/// 在当前插入位置插入elt，将插入位置放在它的后面
pub fn insert(&mut self, elt: T) {
    if self.gap.len() == 0 {
        self.enlarge_gap();
    }

    unsafe {
        let index = self.gap.start;
        std::ptr::write(self.space_mut(index), elt);
    }
    self.gap.start += 1;
}

/// 在当前插入位置插入iter生成的元素，将插入位置放在它们的后面
pub fn insert_iter<I>(&mut self, iterable: I)
    where I: IntoIterator<Item=T>
{
    for item in iterable {
        self.insert(item)
    }
}

/// 删除并返回位于插入位置之后的元素，
/// 如果插入位置在GapBuffer的末尾则返回None
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
    Some(element)
}
```

与 Vec 使用 std::ptr::write 实现推入和使用 std::ptr::read 实现弹出一样，GapBuffer 也使用 write 实现 insert，使用 read 实现 remove。正如 Vec 必须调整其长度以维护初始化的元素与空闲容量之间的界限，GapBuffer 会调用它的间隙。

在间隙被填满时，insert 方法必须增大这个缓冲区以取得更多空闲空间。（impl 块最后的）enlarge_gap 方法负责处理这个：

```
/// 将self.storage的容量增大一倍
fn enlarge_gap(&mut self) {
    let mut new_capacity = self.capacity() * 2;
    if new_capacity == 0 {
        // 当前向量是空的
        // 选择一个合理的起始容量
        new_capacity = 4;
    }

    // 我们不知道缩放Vec对其“未用”容量的影响，因此就简单地
    // 创建了一个新向量并转移了元素
    let mut new = Vec::with_capacity(new_capacity);
    let after_gap = self.capacity() - self.gap.end;
    let new_gap = self.gap.start .. new.capacity() - after_gap;

    unsafe {
        // 转移位于间隙之前的元素
        std::ptr::copy_nonoverlapping(self.space(0),
                                      new.as_mut_ptr(),
                                      self.gap.start);

        // 转移位于间隙之后的元素
        let new_gap_end = new.as_mut_ptr().offset(new_gap.end as isize);
        std::ptr::copy_nonoverlapping(self.space(self.gap.end),
                                      new_gap_end,
                                      after_gap);
    }
    // 这样会释放旧向量，但不会清除元素，因为这个向量的长度是0
    self.storage = new;
    self.gap = new_gap;
}
```

set_position 必须使用 copy 在间隙中来回移动元素，enlarge_gap 则可以使用 copy_nonoverlapping，因为它是把元素移动到一个全新的缓冲区。

把新向量移动到 self.storage 会清除旧向量。因为向量长度始终为 0，所以旧向量认为自己没有需要清除的元素，于是就直接把缓冲区释放了。更巧妙的是，copy_nonoverlapping 会让源内存变成未初始化，因此旧向量认为现在所有元素都归新向量所有是正确的。

最后需要确保清除 GapBuffer 会清除其所有元素：

```
impl<T> Drop for GapBuffer<T> {
    fn drop(&mut self) {
        unsafe {
            for i in 0 .. self.gap.start {
                std::ptr::drop_in_place(self.space_mut(i));
            }
            for i in self.gap.end .. self.capacity() {
                std::ptr::drop_in_place(self.space_mut(i));
            }
        }
    }
}
```

元素都在间隙的前后，因此需要迭代每个区域并使用 `std::ptr::drop_in_place` 函数清除每个元素。这个 `drop_in_place` 函数是一个实用程序，其行为类似于 `drop(std:: ptr::read(ptr))`，但不会费事地把值转移到调用者（所以才可以在非固定大小类型上使用）。而且就像在 `enlarge_gap` 中一样，当向量 `self.storage` 被清除时，其缓冲区也是真的变成了未初始化。

与本章展示的其他类型一样，`GapBuffer` 保证自己的不变性足以遵守所使用的每个不安全特性的协议。因此它的所有公有方法都不需要标记为不安全。`GapBuffer` 实现了一个安全的接口，而这个接口的功能如果使用安全代码来写是不会这么高效的。

21.7.8　不安全代码中的诧异安全性

在 Rust 中，诧异通常不会导致未定义行为，`panic!` 宏并不是一个不安全特性。但当你决定使用不安全代码时，诧异安全性是必须要考虑的问题。

以上一节中的 `GapBuffer::remove` 方法为例：

```
pub fn remove(&mut self) -> Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }

    let element = unsafe {
        std::ptr::read(self.space(self.gap.end))
    };
    self.gap.end += 1;
    Some(element)
}
```

这里调用 `read` 将位于间隙后面的元素移到了这个缓冲区之外，原来的空间变成了未初始化。好在下一行代码增大了间隙以覆盖这个空间，因此在函数返回的时候，一切如常：所有间隙外的元素都是初始化的，而所有间隙内的元素都是未初始化的。

考虑这样一种情况：在调用 `read` 之后、调整到 `self.gap.end` 之前，代码试图使用一个可能诧异的特性，比如通过索引访问切片。在这两个操作之间突然退出会导致 `GapBuffer` 中有一个未初始化的元素位于间隙之外。下次调用 `remove` 可能会尝试再读取它，即使简单地清除 `GapBuffer` 也会尝试清除它。这两个都是未定义行为，因为它们访问了未初始化内存。

类型的方法在完成自己的任务时短暂放松类型的不变性，然后返回前再把一切恢复正常是不可避免的。这样的方法中出现诧异可能中断这个恢复过程，导致类型处于不一致状态。

如果类型只使用安全代码，那么这种不一致可能会导致类型行为失常，但不会导致未定义行为。但是使用不安全特性的代码经常要指望其不变性满足这些特性的协议。破坏不变性会导致破坏协议，而破坏协议会导致未定义行为。

在使用不安全特性时，必须格外注意检查这些敏感区域，确保它们不会做引起诧异的事。

21.8　外来函数：在Rust中调用C和C++

Rust **外来函数接口**允许 Rust 代码调用用 C 或 C++ 编写的函数。

本节会编写一个与 libgit2 链接的程序，libgit2 是一个操作 Git 版本控制系统的 C 库。我们会先展示如何在 Rust 中直接使用 C 函数，然后再展示如何构建调用 libgit2 的安全接口，这些灵感都来自开源的 git2-rs 包。

假设你熟悉 C 和编译、链接 C 程序的机制。C++ 的使用与此类似。假设你熟悉 Git 版本控制系统。

21.8.1 查找共有数据表示

Rust 与 C 的公因子是机器语言，因此为了预见 Rust 值在 C 代码中的表示，或者相反，需要考虑它们的机器级表示。本书从始至终一直强调值在内存中的实际表示，因此你可能已经注意到 C 和 Rust 在内存中的数据表示有很多共性。比如，Rust 的 usize 和 C 的 size_t 完全相同，而结构体在两种语言中基本上也是一回事。为建立 Rust 与 C 类型的对应关系，可以先从原始类型开始，然后再过渡到更复杂的类型。

作为一门系统编程语言，C 对其类型的表示一直非常宽松。比如，int 通常是 32 位长，但更长也可以，更短比如 16 位也行。再比如，char 可以有符号也可以无符号。为了对应这种变化，Rust 的 std::os::raw 模块定义了一组 Rust 类型，保证与某些 C 类型具有相同的表示。这些主要包括原始整数和字符类型，如表 21-1 所示。

表21-1：类型对照

C类型	对应的std::os::raw类型
short	c_short
int	c_int
long	c_long
long long	c_longlong
unsigned short	c_ushort
unsigned、unsigned int	c_uint
unsigned long	c_ulong
unsigned long long	c_ulonglong
char	c_char
signed char	c_schar
unsigned char	c_uchar
float	c_float
double	c_double
void *、const void *	*mut c_void、*const c_void

关于这个表，需要注意以下几点。

- 除了 c_void，这里所有的 Rust 类型都是某个原始 Rust 类型的别名，比如 c_char 是 i8 或 u8。
- 没有被认可的 Rust 类型对应于 C 的 bool。目前，Rust 的 bool 始终要么是 0 要么是一个字节，所有主流 C 和 C++ 实现的表示也相同。不过，Rust 语言团队并未承诺将来会保持这个表示不变，因为这样意味着减少优化的可能性。

- Rust 的 32 位 char 类型并不对应 wchar_t，其宽度和编码因实现而不同。C 的 char32_t 类型接近，但它的编码仍然不保证是 Unicode。
- Rust 的原始 usize 和 isize 类型与 C 的 size_t 和 ptrdiff_t 的表示相同。
- C 和 C++ 指针与 C++ 引用对应 Rust 的原始指针类型 *mut T 和 *const T。
- 严格来讲，C 标准允许实现使用 Rust 没有对应类型的表示，比如 36 位整数、有符号值的符号及大小（sign-and-magnitude）表示等。实践中，在每个 Rust 已经移植到的平台上，每个常用的 C 整数类型在 Rust 中都可以找到对应的类型，bool 除外。

要定义兼容 C 结构体的 Rust 结构体类型，可以使用 #[repr(C)] 属性。把 #[repr(C)] 属性放到结构体定义上方，表示让 Rust 在内存中使用与 C 布局其结构体相同的方式来布局这个结构体的字段。例如，libgit2 的 git2/errors.h 头文件定义了下面的 C 结构体，以提供前面报告错误的细节：

```
typedef struct {
    char *message;
    int klass;
} git_error;
```

可以像下面这样定义一个具有相同表示的 Rust 类型：

```
#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}
```

#[repr(C)] 属性只影响结构体本身的布局，不影响其个别字段，因此要匹配 C 结构体，每个字段也都必须使用类 C 的类型：*const c_char 对应 char *、c_int 对应 int，等等。

在这个特例中，#[repr(C)] 属性可能不会改变 git_error 的布局。指针和整数的布局方式并不太多。但 C 和 C++ 保证结构体的成员以声明它们的顺序出现在内存中，且每个成员都有不同的地址，而 Rust 为了减少结构体占用的内存会对字段重新排序，没有大小的类型不占空间。这里的 #[repr(C)] 属性告诉 Rust 对给定类型遵守 C 的规则。

同样也可以使用 #[repr(C)] 控制 C 式枚举的表示：

```
#[repr(C)]
enum git_error_code {
    GIT_OK         =  0,
    GIT_ERROR      = -1,
    GIT_ENOTFOUND  = -3,
    GIT_EEXISTS    = -4,
     ...
}
```

正常情况下，Rust 在选择如何表示枚举时有很多种花招儿。比如，前面提到 Rust 会使用一个字（如果 T 是固定大小的）来存储 Option<&T>。如果没有 #[repr(C)]，Rust 则会使用一个字节表示这个 git_error_code 枚举。但有了 #[repr(C)]，Rust 就会像 C 那样，使用 C 的 int 大小的一个值。

此外，也可以让 Rust 像表示某个整数类型一样表示枚举。如果前面的定义使用的是 `#[repr(i16)]`，就会得到与以下 C++ 枚举相同的一个 16 位类型的表示：

```
#include <stdint.h>

enum git_error_code: int16_t {
    GIT_OK         =  0,
    GIT_ERROR      = -1,
    GIT_ENOTFOUND  = -3,
    GIT_EEXISTS    = -4,
    ...
};
```

在 Rust 和 C 之间传递字符串要困难一些。C 将字符串表示为一个指向字符数组的指针，以一个空字符终结。Rust 则显式地存储字符串的长度，或是保存在 `String` 的一个字段中，或是作为胖指针 `&str` 的第二个字。Rust 字符串不是以空字符终结的，但其内容中有可能包含空字符，就像其他字符一样。

这意味着不能把 Rust 字符串借用为 C 字符串。如果把一个指向 Rust 字符串的指针传给 C 代码，那它可能会将嵌入的空字符误认为字符串的末尾，或者跑到字符串末尾却找不到空字符。反过来，则可以将 C 字符串指针借用为 Rust 的 `&str`，只要其内容是格式正确的 UTF-8 即可。

这种情况实际上强制 Rust 把 C 字符串看成一种完全不同于 `String` 和 `&str` 的类型。在 `std::ffi` 模块中，`CString` 和 `CStr` 类型分别表示所有型和借用型空字节结尾的字节数组。相比于 `String` 和 `&str`，`CString` 和 `CStr` 的方法非常有限，仅限于构建和转换到其他类型。下一节将介绍这两个类型。

21.8.2　声明外来函数和变量

在 `extern` 块中可以声明由其他库定义的函数或变量，但最终 Rust 可执行文件会链接到它们。例如，每个 Rust 程序都会链接到标准 C 库，因此可以像下面这样在 Rust 中声明 C 库的 `strlen` 函数：

```
use std::os::raw::c_char;

extern {
    fn strlen(s: *const c_char) -> usize;
}
```

这样就告知了 Rust 这个函数的名字和类型，至于函数定义以后再链接。

Rust 假设在 `extern` 块中声明的函数基于 C 的惯例传参并接收返回值。这些函数被定义为 `unsafe` 函数。对于 `strlen` 来说这是正确的：它确实是一个 C 函数，而且其 C 规范要求传入一个有效的指针，指向正确结尾的字符串，而这是 Rust 无法强制的协议。（几乎任何以原始指针为参数的函数都必须是 `unsafe`：安全 Rust 可以基于任意整数构建原始指针，而解引用这样的指针是未定义行为。）

有了 `extern` 块，就可以像调用其他 Rust 函数一样调用 `strlen` 了，当然类型还是暴露了它

外来者的身份：

```
use std::ffi::CString;

let rust_str = "I'll be back";
let null_terminated = CString::new(rust_str).unwrap();
unsafe {
    assert_eq!(strlen(null_terminated.as_ptr()), 12);
}
```

`CString::new` 函数构建了一个空字符结尾的 C 字符串。它首先会检查参数是否嵌入了空字符，因为嵌入的空字符不能在 C 字符串中表示，如果发现有则返回错误（这也是对结果调用 `unwrap` 的原因）。否则，它会在字符串末尾添加空字节，返回一个拥有该结果字符串的 `CString`。

`CString::new` 的开销取决于你传入的是什么类型。它可以接收实现 `Into<Vec<u8>>` 的任何值。传入 `&str` 需要一次分配和一次复制，因为转换为 `Vec<u8>` 时要构建这个字符串的一个分配在堆上的副本，以便向量拥有它。而传入 `String` 的值只需要消费该字符串，接管其缓冲区，因此除非追加空字符会强制缓冲区扩张，否则转换根本不需要文本复制和内存分配。

`CString` 会解引用为 `CStr`，其 `as_ptr` 方法会返回一个指向字符串开头的 `*const c_char` 指针。这个类型是 `strlen` 所需要的。在这个例子中，`strlen` 会扫描字符串，查找 `CString::new` 放到末尾的空字符，然后返回长度，即字节数。

在 `extern` 块中还可以声明全局变量。POSIX 系统有一个名为 `environ` 的全局变量，其保存进程的环境变量值。在 C 中，它是这样声明的：

```
extern char **environ;
```

在 Rust 中，则需要这样写：

```
use std::ffi::CStr;
use std::os::raw::c_char;

extern {
    static environ: *mut *mut c_char;
}
```

要打印这个环境变量的第一个元素，要这样做：

```
unsafe {
    if !environ.is_null() && !(*environ).is_null() {
        let var = CStr::from_ptr(*environ);
        println!("first environment variable: {}",
                 var.to_string_lossy())
    }
}
```

在确定 `environ` 至少有一个元素后，代码调用了 `CStr::from_ptr` 来构建借用它的 `CStr`。`to_string_lossy` 方法返回了一个 `Cow<str>`：如果这个 C 字符串包含格式正确的 UTF-8，`Cow` 就将其内容借用为 `&str`，不包含末尾的空字节；否则，`to_string_lossy` 在堆里生成相

应文本的副本，将其中不合法的 UTF-8 序列替换为正式的 Unicode 替代字符 '�'，并为其构建一个所有型 Cow。不管怎样，结果都会实现 Display，因此可以使用格式化参数 {} 把它打印出来。

21.8.3 使用库函数

要使用某个库提供的函数，可以在 extern 块上方放一个 #[link] 属性，标示出 Rust 需要链接到可执行文件的库名。例如，下面这个程序调用了 libgit2 的初始化和停止方法，没干别的事：

```
use std::os::raw::c_int;

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
}

fn main() {
    unsafe {
        git_libgit2_init();
        git_libgit2_shutdown();
    }
}
```

这里像以前一样通过 extern 块声明了外部函数。把 #[link(name ="git2")] 属性放到包里的意思是说，Rust 在创建最终可执行文件或共享库时，应该链接 git2 库。Rust 使用系统链接器构建可执行文件。在 Unix 上，它会在链接器命令行上传入 -lgit2 参数；在 Windows 上，它会传入 git2.LIB。

#[link] 属性也可以用于库包。在构建依赖其他包的程序时，Cargo 会从整个依赖图中提取这些链接说明，并将它们全部包含到最终链接中。

对这个例子而言，如果你想在自己的机器上照着做，还需要构建 libgit2。我们使用了 libgit2 的 0.25.1 版。要编译 libgit2，则需要安装 CMake 构建工具和 Python 语言。我们使用的是 CMake 3.8.0 和 Python 2.7.13。

构建 libgit2 的完整说明可以在其网站找到，其实很简单，这里将进行演示。在 Linux 上，假设你已经把这个库的源代码解压缩到了目录 /home/jimb/libgit2-0.25.1 中：

```
$ cd /home/jimb/libgit2-0.25.1
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

在 Linux 上，这样会生成一个共享库 /home/jimb/libgit2-0.25.1/build/libgit2.so.0.25.1，以及指向它的一批符号链接，其中一个名为 libgit2.so。在 macOS 上，结果类似，但这个库名为 libgit2.dylib。

在 Windows 上，构建也很简单。假设你已经把源代码解压缩到了目录 C:\Users\JimB\libgit2-0.25.1 中。在 Visual Studio 的命令行中执行：

```
> cd C:\Users\JimB\libgit2-0.25.1
> mkdir build
> cd build
> cmake -A x64 ..
> cmake --build .
```

这些命令跟 Linux 上是一样的，只是在第一次运行 CMake 时必须指明要 64 位构建，这样才能与 Rust 编译器匹配。（如果你安装的是 32 位 Rust 工具链，那么在第一个 `cmake` 命令后面可以省略 `-A x64` 标记。）这样可以生成一个导入库 git2.LIB 和一个动态链接库 git2.DLL，二者都在目录 C:\Users\JimB\libgit2-0.25.1\build\Debug 中。（其他构建说明针对的是 Unix，不适用于 Windows。）

在另外一个目录中创建 Rust 程序：

```
$ cd /home/jimb
$ cargo new --bin git-toy
```

把上面的代码放到 src/main.rs 中。当然，如果你现在尝试构建，Rust 则不知道去哪里找 `libgit2`：

```
$ cd git-toy
$ cargo run
   Compiling git-toy v0.1.0 (file:///home/jimb/git-toy)
error: linking with `cc` failed: exit code: 1
  |
  = note: "cc" ... "-l" "git2" ...
  = note: /usr/bin/ld: cannot find -lgit2
          collect2: error: ld returned 1 exit status


error: aborting due to previous error

error: Could not compile `git-toy`.

To learn more, run the command again with --verbose.
$
```

为此需要写一个**构建脚本**告诉 Rust 到哪里搜索库，也就是 Cargo 在构建时要编译和运行的 Rust 代码。构建脚本可以做任何事，包括动态生成代码、编译要包含在包中的 C 代码等。对目前而言，我们需要给可执行文件的链接命令添加一个库搜索路径。Cargo 在运行这个构建脚本时，会解析构建脚本的输出，从中获取这个信息。因此构建脚本只要把相关命令和路径打印到标准输出即可。

要创建构建脚本，需要在 Cargo.toml 文件所在目录下添加一个名为 build.rs 的文件，包含以下内容：

```
fn main() {
    println!(r"cargo:rustc-link-search=native=/home/jimb/libgit2-0.25.1/build");
}
```

这是 Linux 的正确路径。在 Windows 上，需要把 native= 后面的内容改为 C:\Users\JimB\libgit2-0.25.1\build\Debug。（这里为简单起见，省了仪式。在真正的应用中，不应该在构建脚本中使用绝对路径。本节末尾会引用文档的正确做法。）

接下来，告诉 Cargo 这是你的构建脚本。为此，要在 Cargo.toml 文件的 [package] 区域中加上 build = "build.rs"。整个文件内容如下：

```
[package]
name = "git-toy"
version = "0.1.0"
authors = ["You <you@example.com>"]
build = "build.rs"

[dependencies]
```

现在差不多可以运行程序了。在 macOS 上，应该可以立即运行。而在 Linux 系统上，运行后可能会看到如下信息：

```
$ cargo run
   Compiling git-toy v0.1.0 (file:///home/jimb/git-toy)
    Finished dev [unoptimized + debuginfo] target(s) in 0.64 secs
     Running `target/debug/git-toy`
target/debug/git-toy: error while loading shared libraries:
libgit2.so.25: cannot open shared object file: No such file or directory
$
```

这说明虽然 Cargo 成功地在可执行文件中链接了库，但它不知道在运行时如何找到共享库。Windows 对此会弹出一个对话框来显示失败消息。在 Linux 上，必须设置 LD_LIBRARY_PATH 环境变量：

```
$ export LD_LIBRARY_PATH=/home/jimb/libgit2-0.25.1/build:$LD_LIBRARY_PATH
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/git-toy`
$
```

在 macOS 上，则需要设置 DYLD_LIBRARY_PATH。

在 Windows 上，必须设置 PATH 环境变量：

```
> set PATH=C:\Users\JimB\libgit2-0.25.1\build\Debug;%PATH%
> cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
    Running `target/debug/git-toy`
>
```

当然，在已部署的应用中，我们不想为了找到库代码而设置环境变量。这时可以把 C 库静态链接到 Rust 包。这样就会把库的对象文件复制到包的 .rlib 文件中，与包 Rust 代码的对象文件和元数据放在了一起。然后，整个集合将参与最终链接。

Cargo 约定，对提供 C 库访问的包要命名为 LIB-sys 这种形式，其中 LIB 是 C 库的名字，而 -sys 包应该只包含静态链接库和带有 extern 块及类型定义的 Rust 模块。高层级接口相应地归属于依赖这个 -sys 包的包。这样多个上游包可以依赖相同的 -sys 包，假设 -sys 包

的一个版本可以满足它们的所有需求。

关于 Cargo 支持构建脚本和链接系统库的详细信息，请参考 Cargo 在线文档。文档中介绍了如何避免在构建脚本中使用绝对路径、控制编译标记、使用 `pkg-config` 工具等。`git2-rs` 包也是一个用来模拟的好例子，它的构建脚本要处理一些复杂情况。

21.8.4 libgit2的原始接口

正确使用 `libgit2` 首先要明确两个问题。

- 在 Rust 中使用 `libgit2` 函数需要接收什么？
- 如何在这些函数基础之上构建安全的 Rust 接口？

下面分别来回答这两个问题。本节将写一个程序，基本上就是一个大型的 `unsafe` 块，其中全都是不符合 Rust 惯例的代码，反映出了两种语言混合在类型系统和编码惯例上的冲突。我们称这个程序为**原始**接口。代码可能比较乱，但总归把 Rust 代码使用 `libgit2` 的过程都写清楚了。

然后，下一节会构建访问 `libgit2` 的安全接口。届时将让 Rust 类型来强制落实 `libgit2` 对用户给出的规则。好在 `libgit2` 是设计精良的 C 库，因此 Rust 的安全性迫使我们提出的问题都有很好的答案，从而可以构建没有 `unsafe` 函数的符合 Rust 惯例的接口。

本节要写的程序非常简单：通过命令行参数接收一个路径，打开对应的 Git 仓库，把最近一次提交的信息打印出来。但这对于演示如何构建安全和符合 Rust 惯例的接口来说已经足够了。

在这个原始接口中，程序最终用到的 `libgit2` 中的函数和类型比之前要多很多。因此有必要把 `extern` 块转移到一个独立的模块中。为此要在 git-toy/src 中创建一个名为 raw.rs 的文件，其内容如下：

```
#![allow(non_camel_case_types)]
use std::os::raw::{c_int, c_char, c_uchar};

#[link(name = "git2")]
extern {
    pub fn git_libgit2_init() -> c_int;
    pub fn git_libgit2_shutdown() -> c_int;
    pub fn giterr_last() -> *const git_error;

    pub fn git_repository_open(out: *mut *mut git_repository,
                               path: *const c_char) -> c_int;
    pub fn git_repository_free(repo: *mut git_repository);

    pub fn git_reference_name_to_id(out: *mut git_oid,
                                    repo: *mut git_repository,
                                    reference: *const c_char) -> c_int;

    pub fn git_commit_lookup(out: *mut *mut git_commit,
                             repo: *mut git_repository,
                             id: *const git_oid) -> c_int;
```

```
    pub fn git_commit_author(commit: *const git_commit) -> *const git_signature;
    pub fn git_commit_message(commit: *const git_commit) -> *const c_char;
    pub fn git_commit_free(commit: *mut git_commit);
}

pub enum git_repository {}
pub enum git_commit {}

#[repr(C)]
pub struct git_error {
    pub message: *const c_char,
    pub klass: c_int
}

#[repr(C)]
pub struct git_oid {
    pub id: [c_uchar; 20]
}

pub type git_time_t = i64;

#[repr(C)]
pub struct git_time {
    pub time: git_time_t,
    pub offset: c_int
}

#[repr(C)]
pub struct git_signature {
    pub name: *const c_char,
    pub email: *const c_char,
    pub when: git_time
}
```

这里每一项都照搬了 `libgit2` 自己的头文件。例如，libgit2-0.25.1/include/git2/repository.h 包含如下声明：

```
extern int git_repository_open(git_repository **out, const char *path);
```

这个函数尝试打开路径为 `path` 的 Git 仓库。如果一切顺利，它会创建一个 `git_repository` 对象，并在 `out` 指向的位置存储一个指向它的指针。等价的 Rust 声明如下：

```
pub fn git_repository_open(out: *mut *mut git_repository,
                           path: *const c_char) -> c_int;
```

`libgit2` 的公有头文件将 `git_repository` 定义为一个不完整结构体类型的 `typedef`：

```
typedef struct git_repository git_repository;
```

因为这个类型的细节是库私有的，所以公有头文件不会定义 `struct git_repository`，以确保这个库的用户永远不会自己构建该类型的实例。Rust 中与不完整结构体类型对应的定义可以这样：

```
pub enum git_repository {}
```

这是一个没有变体的枚举类型。在 Rust 中无法创建这样一个类型的值。这是一个“怪物”，但它完美映射了只有 libgit2 才可能构建的一种 C 类型，并且只能通过原始指针来操作。

手工写一个大 extern 块是很麻烦的。如果你正在创建一个复杂 C 库的 Rust 接口，可以试一试 bindgen 包。这个包中的函数可以用在构建脚本中解析 C 头文件，并自动生成对应的 Rust 声明。因为篇幅有限，这里就不展示如何使用 bindgen 了，但它在 crates.io 的页面上有相关文档的链接。

接下来要完全重写 main.rs。首先，需要声明 raw 模块：

```
mod raw;
```

根据 libgit2 的约定，不可靠的函数会返回整数，正值或 0 表示成功，负值表示失败。如果发生错误，giterr_last 函数会返回一个指向 git_error 结构体的指针，提供关于错误的更详细信息。libgit2 拥有这个结构体，因此我们不需要自己释放它，不过它可能会被下一次库调用覆盖。真正的 Rust 接口会使用 Result，但在这个原始版中，我们想原封不动地使用 libgit2 函数，因此必须自己写函数来处理错误：

```
use std::ffi::CStr;
use std::os::raw::c_int;

fn check(activity: &'static str, status: c_int) -> c_int {
    if status < 0 {
        unsafe {
            let error = &*raw::giterr_last();
            println!("error while {}: {} ({})",
                     activity,
                     CStr::from_ptr(error.message).to_string_lossy(),
                     error.klass);
            std::process::exit(1);
        }
    }

    status
}
```

我们会使用这个函数检查 libgit2 调用的结果，如下所示：

```
check("initializing library", raw::git_libgit2_init());
```

这里又使用了前面使用过的 CStr 方法：from_ptr 用于从 C 字符串构建 CStr，to_string_lossy 用于将其转换为 Rust 可以打印的值。

接下来需要一个函数打印提交消息：

```
unsafe fn show_commit(commit: *const raw::git_commit) {
    let author = raw::git_commit_author(commit);

    let name = CStr::from_ptr((*author).name).to_string_lossy();
    let email = CStr::from_ptr((*author).email).to_string_lossy();
    println!("{} <{}>\n", name, email);

    let message = raw::git_commit_message(commit);
    println!("{}", CStr::from_ptr(message).to_string_lossy());
}
```

拿到指向 git_commit 的指针后，show_commit 调用 git_commit_author 和 git_commit_message 取得所需的信息。这两个函数遵循 libgit2 文档中给出的约定：

> 如果函数的返回值是一个对象，那这个函数就是一个获取函数，而该对象的生命期附属于父对象。

用 Rust 的话说，author 和 message 都是从 commit 借用的，也就是说，show_commit 不需要自己释放它们。但是，它也不能在 commit 被释放以后还持有它们。因为这个 API 使用原始指针，所以 Rust 不会为我们检查它们的生命期。如果真的意外创建了悬空指针，那恐怕要到程序崩溃时才会发现。

前面的代码假设这些字段保存 UTF-8 文本，但这并不总是对的。Git 也允许其他编码。正确解释这些字符串可能需要使用 encoding 包。为简单起见，就先不考虑这些问题了。

我们程序的 main 函数如下：

```
use std::ffi::CString;
use std::mem;
use std::ptr;
use std::os::raw::c_char;

fn main() {
    let path = std::env::args().skip(1).next()
        .expect("usage: git-toy PATH");
    let path = CString::new(path)
        .expect("path contains null characters");

    unsafe {
        check("initializing library", raw::git_libgit2_init());

        let mut repo = ptr::null_mut();
        check("opening repository",
              raw::git_repository_open(&mut repo, path.as_ptr()));

        let c_name = b"HEAD\0".as_ptr() as *const c_char;
        let mut oid = mem::uninitialized();
        check("looking up HEAD",
              raw::git_reference_name_to_id(&mut oid, repo, c_name));

        let mut commit = ptr::null_mut();
        check("looking up commit",
              raw::git_commit_lookup(&mut commit, repo, &oid));

        show_commit(commit);

        raw::git_commit_free(commit);

        raw::git_repository_free(repo);

        check("shutting down library", raw::git_libgit2_shutdown());
    }
}
```

代码首先处理路径参数并初始化库，这些前面都见过。第一段新代码是这样的：

```
let mut repo = ptr::null_mut();
check("opening repository",
      raw::git_repository_open(&mut repo, path.as_ptr()));
```

调用 `git_repository_open` 会尝试打开给定路径的 Git 仓库。如果成功，则会为它分配一个新的 `git_repository` 对象，并设置让 `repo` 指向它。Rust 隐式地将引用转换为原始指针，因此这里传入 `&mut repo` 提供了调用所需的 `*mut *mut git_repository`。

这展示了用到的另一个 `libgit2` 约定。同样，摘自 `libgit2` 文档：

> 通过第一个参数以指针对指针形式返回的对象由调用者所有，并负责释放它们。

用 Rust 的话说，就是函数 `git_repository_open` 会把新值的所有权传给调用者。

接下来，再看看查找仓库最新提交对象散列的代码：

```
let mut oid = mem::uninitialized();
check("looking up HEAD",
      raw::git_reference_name_to_id(&mut oid, repo, c_name));
```

`git_oid` 类型存储一个对象标识符，这是 Git 在内部（及令人愉快的用户界面中）使用的一个 160 位的散列码，用于标识提交和文件的各个版本等。这里调用 `git_reference_name_to_id` 查找的是当前 `"HEAD"` 提交的对象标识符。

在 C 中，通过把指针传给函数并由函数来初始化变量是很常见的。`git_reference_name_to_id` 函数在这里就是将它的第一个参数当作一个要初始化的变量。但 Rust 不允许借用一个未初始化变量的引用。可以用 0 来初始化 `oid`，但这是一种浪费，因为这时候存储的任何值都会被覆盖。

把 `oid` 初始化为 `uninitialized()` 可以解决这个问题。这里的 `std::mem::uninitialized` 函数返回了一个你想要的任意类型的值，只不过这个值包含的都是未初始化的位，并且也没有机器码实际用于生成这个值。不过，Rust 在这里会认为 `oid` 已经被赋予了某种值，因此允许我们借用对它的引用。可以想象，从一般意义上说，这是非常不安全的。读取未初始化的值是未定义行为，如果这个值的某部分实现了 `Drop`，那即使清除它也是未定义行为。只能做一些安全的事情。

- 通过 `std::ptr::write` 覆盖它，因为 `std::ptr::write` 要求目标是未初始化的。
- 把它传给 `std::mem::forget`，因为 `std::mem::forget` 会取得其第一个参数的所有权，并且不清除它就能让它消失（对初始化的值这么做会导致内存泄漏）。
- 把它传给一个可以初始化它的外来函数，比如 `git_reference_name_to_id`。

如果调用成功，`oid` 就变成真正初始化了，一切正常。如果调用失败，那么函数不会使用 `oid`，其类型并不需要被清除，因此代码在这种情况下也安全。

对 `repo` 和 `commit` 变量也可以应用 `uninitialized`，但由于它们只有一个字长，而且 `uninitialized` 使用起来有危险，因此干脆直接把它们初始化为空：

```
let mut commit = ptr::null_mut();
check("looking up commit",
      raw::git_commit_lookup(&mut commit, repo, &oid));
```

这样会取得提交对象的标识符并查找实际的提交，成功后会将一个 `git_commit` 指针保存在 `commit` 中。

`main` 函数剩下的代码就都一目了然了。它调用前面定义的 `show_commit` 函数，释放提交和仓库对象，最后关闭库。

现在可以拿手头任何一个现成的 Git 仓库来试一下程序：

```
$ cargo run /home/jimb/rbattle
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/git-toy /home/jimb/rbattle`
Jim Blandy <jimb@red-bean.com>

Animate goop a bit.

$
```

21.8.5 libgit2的安全接口

`libgit2` 的原始接口完美展示了一种不安全特性：尽管可以正确使用（正如前面所做的），但 Rust 不能强制你必须遵守规则。给这样一个库设计安全 API 涉及找到所有规则，然后想办法将违反这些规则的操作都用一种类型或借用检查错误来表示。

下面就是程序用到的 `libgit2` 特性的规则。

- 在使用任何库函数之前必须调用 `git_libgit2_init`。在调用 `git_libgit2_shutdown` 之后，不能使用任何库函数。
- 除输出参数，传给 `libgit2` 函数的所有值都必须完全初始化。
- 调用失败时，传入的要保存调用结果的输出参数保持未初始化，不能使用它们的值。
- `git_commit` 对象会引用派生它的 `git_repository` 对象，因此前者的生命期不能超过后者。（`libgit2` 文档并未载明这些，这是由接口中某些函数的存在推断出并通过阅读源码验证的。）
- 类似地，`git_signature` 始终借用自给定的 `git_commit`，因此前者的生命期也不能超过后者。（文档里确实提到了这一点。）
- 与提交关联的消息以及作者的姓名、电子邮箱地址都是从提交借用的，在提交释放后不能再使用。
- `libgit2` 对象一旦被释放，就不能再使用。

实际上，可以基于 `libgit2` 构建能够强制执行这些规则的 Rust 接口，要么通过 Rust 的类型系统，要么通过内部处理细节。

开始之前，先重新规划一下项目结构。我们希望有一个 `git` 模块暴露这个安全接口，而前面程序中的原始接口将作为它的私有子模块。

整个源代码树的结构如下：

```
git-toy/
├── Cargo.toml
├── build.rs
└── src/
    ├── main.rs
    └── git/
        ├── mod.rs
        └── raw.rs
```

按照 8.2.1 节讲解的规则，`git` 模块的源代码放在 git/mod.rs 中，它的 `git::raw` 子模块的源代码则放在 git/raw.rs 中。

同样，这一次又要完全重写 main.rs。第一行代码要声明 `git` 模块：

```
mod git;
```

然后，需要创建 `git` 子目录，把 raw.rs 移过去：

```
$ cd /home/jimb/git-toy
$ mkdir src/git
$ mv src/raw.rs src/git/raw.rs
```

`git` 模块需要声明自己的 `raw` 子模块。文件 src/git/mod.rs 必须标明如下内容：

```
mod raw;
```

因为它不是 `pub`，所以这个子模块对主程序不可见。

稍后需要使用 `libc` 包中的一些函数，因此必须在 Cargo.toml 中添加依赖项。完整的文件如下所示：

```
[package]
name = "git-toy"
version = "0.1.0"
authors = ["Jim Blandy <jimb@red-bean.com>"]
build = "build.rs"

[dependencies]
libc = "0.2.23"
```

对应的 `extern crate` 项必须出现在 src/main.rs 中：

```
extern crate libc;
```

它需要使用此错误类型的自己的结果类型。

模块已经重新安排就位，接下来看错误处理。即便是 `libgit2` 的初始化函数也可以返回一个错误码，因此开始之前需要先对其分类。惯常的 Rust 接口需要自己的 `Error` 类型从 `giterr_last` 捕获 `libgit2` 的失败码，以及错误消息和分类。适当的错误类型必须实现惯常的 `Error`、`Debug` 和 `Display` 特型。然后，它需要自己拥有的 `Result` 类型使用这个 `Error` 类型。下面是 src/git/mod.rs 中必要的定义：

```
use std::error;
use std::fmt;
use std::result;
```

```
#[derive(Debug)]
pub struct Error {
    code: i32,
    message: String,
    class: i32
}

impl fmt::Display for Error {
    fn fmt(&self, f: &mut fmt::Formatter) -> result::Result<(), fmt::Error> {
        // 显示Error就是把libgit2返回的消息显示出来
        self.message.fmt(f)
    }
}

impl error::Error for Error {
    fn description(&self) -> &str { &self.message }
}

pub type Result<T> = result::Result<T, Error>;
```

为检查调用原始库的结果，模块需要定义一个把返回的错误码转换为 Result 的函数：

```
use std::os::raw::c_int;
use std::ffi::CStr;

fn check(code: c_int) -> Result<c_int> {
    if code >= 0 {
        return Ok(code);
    }

    unsafe {
        let error = raw::giterr_last();

        // libgit2保证(*error).message始终不为空，而且以空字节结尾，
        // 因此这个调用是安全的
        let message = CStr::from_ptr((*error).message)
            .to_string_lossy()
            .into_owned();

        Err(Error {
            code: code as i32,
            message,
            class: (*error).klass as i32
        })
    }
}
```

这个 check 函数与原始接口中 check 函数的主要区别是它构建了一个 Error 值，而不是打印错误消息后立即退出。

现在可以开始考虑 libgit2 初始化了。这个安全接口会提供一个 Repository 类型，表示打开的 Git 仓库，并包含可以解析引用、查找提交等的方法。下面是 Repository 在 git/mod.rs 中的定义：

```
/// 一个Git仓库
pub struct Repository {
    // 这必须始终是一个指向活的git_repository结构体的指针
    // 其他Repository都不能再指向它
    raw: *mut raw::git_repository
}
```

Repository 的 raw 字段不是公有的。因为只有这个模块中的代码可以访问 raw::git_repository 指针。要让模块安全，必须保证这个指针始终能够得到正确使用。

如果创建 Repository 的唯一方式是成功打开一个新 Git 仓库，那就可以确保每个 Repository 都指向不同的 git_repository 对象。

```
use std::path::Path;

impl Repository {
    pub fn open<P: AsRef<Path>>(path: P) -> Result<Repository> {
        ensure_initialized();

        let path = path_to_cstring(path.as_ref())?;
        let mut repo = null_mut();
        unsafe {
            check(raw::git_repository_open(&mut repo, path.as_ptr()))?;
        }
        Ok(Repository { raw: repo })
    }
}
```

因为要使用这个安全接口必须先取得一个 Repository 值，而 Repository::open 一上来就调用了 ensure_initialized，所以可以确信 ensure_initialized 会在任何 libgit2 函数之前被调用。这个函数的定义如下：

```
use std;
use libc;

fn ensure_initialized() {
    static ONCE: std::sync::Once = std::sync::ONCE_INIT;
    ONCE.call_once(|| {
        unsafe {
            check(raw::git_libgit2_init())
                .expect("initializing libgit2 failed");
            assert_eq!(libc::atexit(shutdown), 0);
        }
    });
}

use std::io::Write;

extern fn shutdown() {
    unsafe {
        if let Err(e) = check(raw::git_libgit2_shutdown()) {
            let _ = writeln!(std::io::stderr(),
                             "shutting down libgit2 failed: {}",
                             e);
```

```
            std::process::abort();
        }
    }
}
```

std::sync::Once 类型帮助我们以线程安全的方式来运行初始化代码。只有第一个调用 ONCE.call_once 的线程会运行给定的闭包。后续的任何调用，无论通过这个线程还是其他线程，都会阻塞到第一个调用完成，然后立即返回，不会再次运行这个闭包。闭包执行完毕再调用 ONCE.call_once 开销很低，只涉及对保存在 ONCE 中一个标志位的一次原子加载。

在前面的代码中，初始化闭包调用 git_libgit2_init 并检查了结果。它偷了一点懒，只用 expect 来确保初始化成功，而没有把错误传播给调用者。

为确保程序调用 git_libgit2_shutdown，这个初始化闭包使用了 C 库的 atexit 函数。这个函数接收一个在退出进程前要调用函数的指针。Rust 闭包不能作为 C 函数指针来用，闭包是某种匿名类型的值，包含它捕获的任何变量的值或引用。而 C 函数指针只是一个指针。不过，可以使用 Rust 的 fn 类型，只要把它们声明为 extern 以便 Rust 知道使用 C 调用惯例即可。局部函数 shutdown 刚好满足要求，能保证 libgit2 正常关闭。

7.1.1 节曾提到过跨语言诧异是未定义行为。atexit 调用 shutdown 就越过了这个边界，因此关键在于 shutdown 不要诧异。这也是 shutdown 不能再简单地使用 expect 处理 raw::git_libgit2_shutdown 返回错误的原因。此时，它必须上报错误并自己终止进程。POSIX 禁止在 atexit 处理程序中调用 exit，因此 shutdown 调用 std::process::abort 硬性终止程序。

适当地提前调用 git_libgit2_shutdown 是有可能的，比如在最后一个 Repository 值被清除时。但无论怎么调整，调用 git_libgit2_shutdown 都必须是这个安全 API 的责任。从调用它的那一刻起，任何现存的 libgit2 对象都会变得不能安全使用，因此安全 API 必须不直接暴露这个函数。

Repository 的原始指针必须始终指向一个活的 git_repository 对象。这意味着关闭一个仓库的唯一方式是清除拥有该对象的 Repository 值：

```
impl Drop for Repository {
    fn drop(&mut self) {
        unsafe {
            raw::git_repository_free(self.raw);
        }
    }
}
```

通过只在指向 raw::git_repository 的唯一指针要被释放时调用 git_repository_free，Repository 类型也保证了指针在释放后永远不会再被使用。

Repository::open 方法使用了一个名为 path_to_cstring 的私有方法，该方法有两个定义：一个针对类 Unix 系统，一个针对 Windows：

```
use std::ffi::CString;

#[cfg(unix)]
fn path_to_cstring(path: &Path) -> Result<CString> {
```

```
    // as_bytes方法只存在于类Unix系统中
    use std::os::unix::ffi::OsStrExt;

    Ok(CString::new(path.as_os_str().as_bytes())?)
}

#[cfg(windows)]
fn path_to_cstring(path: &Path) -> Result<CString> {
    // 尝试转换到UTF-8。如果失败，则libgit2也不能处理这个路径
    match path.to_str() {
        Some(s) => Ok(CString::new(s)?),
        None => {
            let message = format!("Couldn't convert path '{}' to UTF-8",
                                  path.display());
            Err(message.into())
        }
    }
}
```

这个 `libgit2` 接口让代码变得有点棘手。在所有平台上，`libgit2` 都接收空字节结尾的 C 字符串作为路径。在 Windows 上，`libgit2` 假设这些 C 字符串包含格式正确的 UTF-8 并内部将它们转换为 Windows 实际要求的 16 位路径。通常情况下这样没问题，但并不理想。Windows 允许文件名包含格式不正确的 Unicode，因此就不能转换为 UTF-8。如果遇到这样的文件，不可能把它的名字传给 `libgit2`。

在 Rust 中，正确的文件系统路径是一个 `std::path::Path`。这个类型经过精心设计，可以处理 Windows 或 POSIX 系统中的任何合法路径。这意味着在 Windows 上有一些 `Path` 值不能传给 `libgit2`，因为它们不是格式正确的 UTF-8。因此尽管 `path_to_cstring` 的行为不尽如人意，但已经是基于 `libgit2` 接口所能给出的最好方案了。

刚刚看到的这两个 `path_to_cstring` 定义依赖我们的 `Error` 类型转换，其中的 `?` 操作符尝试进行这种转换，Windows 版则显式地调用 `.into()`。这些转换没什么可说的：

```
impl From<String> for Error {
    fn from(message: String) -> Error {
        Error { code: -1, message, class: 0 }
    }
}

// NulError是CString::new在字符串包含嵌入的空字节时返回的错误
impl From<std::ffi::NulError> for Error {
    fn from(e: std::ffi::NulError) -> Error {
        Error { code: -1, message: e.to_string(), class: 0 }
    }
}
```

接下来要想办法把 `Git` 引用解析为一个对象标识符。因为对象标识符不过就是 20 字节的散列值，所以在安全 API 中暴露出来完全没问题：

```
/// 标识符，代表保存在Git对象数据库中的某种对象，
/// 比如提交、树、大文件、标签，等等。是对象
/// 内容的长散列
pub struct Oid {
```

```
        pub raw: raw::git_oid
    }
```

我们要给 Repository 添加一个方法来执行查找：

```
    use std::mem::uninitialized;
    use std::os::raw::c_char;

    impl Repository {
        pub fn reference_name_to_id(&self, name: &str) -> Result<Oid> {
            let name = CString::new(name)?;
            unsafe {
                let mut oid = uninitialized();
                check(raw::git_reference_name_to_id(&mut oid, self.raw,
                                                    name.as_ptr() as *const c_char))?;
                Ok(Oid { raw: oid })
            }
        }
    }
```

尽管 oid 在查找失败时会保持未初始化，这个函数也可以保证调用者永远看不到这个未初始化的值。很简单，它只是遵循了 Rust 的 Result 惯例：调用者得到的要么是一个 Ok，要么是一个 Err，其中前者包含已经适当初始化的 Oid 值。

接下来，模块需要一种从仓库中取得提交的方式。下面是我们定义的 Commit 类型：

```
    use std::marker::PhantomData;

    pub struct Commit<'repo> {
        // 必须始终是一个指向可用git_commit结构体的指针
        raw: *mut raw::git_commit,
        _marker: PhantomData<&'repo Repository>
    }
```

如前所述，git_commit 对象的生命期不能超过从中取得它的 git_repository 对象。Rust 的生命期让代码精确地表达了这个规则。

在本章前面 RefWithFlag 的例子中，我们曾使用 PhantomData 字段告诉 Rust 视同一个类型包含给定生命期的某个引用，而实际上该类型明显不包含这个引用。Commit 类型也需要做类似处理。这里，_marker 字段的类型是 PhantomData<&'repo Repository>，表明 Rust 应该将 Commit<'repo> 当成就好像它包含生命期为 'repo 的某个 Repository 的引用。

查找提交的方法如下所示：

```
    use std::ptr::null_mut;

    impl Repository {
        pub fn find_commit(&self, oid: &Oid) -> Result<Commit> {
            let mut commit = null_mut();
            unsafe {
                check(raw::git_commit_lookup(&mut commit, self.raw, &oid.raw))?;
            }
            Ok(Commit { raw: commit, _marker: PhantomData })
        }
    }
```

这里是如何把 Commit 的生命期关联到 Repository 的？按照 5.2.7 节介绍过的规则，find_commit 方法的签名省略了相关引用的生命期。如果把生命期写出来，完整的签名应该如下所示：

```
fn find_commit<'repo, 'id>(&'repo self, oid: &'id Oid)
    -> Result<Commit<'repo>>
```

这正是我们想要的：Rust 视同返回的 Commit 借用了 self，也就是 Repository。

当 Commit 被清除时，它必须释放自己引用的 raw::git_commit：

```
impl<'repo> Drop for Commit<'repo> {
    fn drop(&mut self) {
        unsafe {
            raw::git_commit_free(self.raw);
        }
    }
}
```

可以从 Commit 借用一个 Signature（姓名、电子邮箱地址）和提交消息的文本：

```
impl<'repo> Commit<'repo> {
    pub fn author(&self) -> Signature {
        unsafe {
            Signature {
                raw: raw::git_commit_author(self.raw),
                _marker: PhantomData
            }
        }
    }

    pub fn message(&self) -> Option<&str> {
        unsafe {
            let message = raw::git_commit_message(self.raw);
            char_ptr_to_str(self, message)
        }
    }
}
```

Signature 类型的定义如下：

```
pub struct Signature<'text> {
    raw: *const raw::git_signature,
    _marker: PhantomData<&'text str>
}
```

git_signature 对象总会从别的地方借用自己的文本。特别地，git_commit_author 返回的签名从 git_commit 借用自己的文本。因此我们的安全的 Signature 类型包含一个 PhantomData<&'text str>，以告诉 Rust 视同该类型包含一个生命期为 'text 的 &str。跟以前一样，什么也不用多写，Commit::author 就可以正确地将它返回的 Signature 的 'text 生命期与 Commit 的生命期关联起来。而 Commit::message 方法对 Option<&str> 包含的提交消息也进行了同样的关联。

Signature 包含取得作者姓名和电子邮箱地址的方法：

```
impl<'text> Signature<'text> {
    /// 将作者姓名返回为&str，或者如果不是格式正确的UTF-8则返回None
    pub fn name(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).name)
        }
    }

    /// 将作者的电子邮箱地址返回为&str，或者如果不是格式正确的UTF-8则返回None
    pub fn email(&self) -> Option<&str> {
        unsafe {
            char_ptr_to_str(self, (*self.raw).email)
        }
    }
}
```

前面这些方法都依赖一个私有辅助函数 `char_ptr_to_str`：

```
/// 尝试从ptr借用一个&str，ptr可能为空，也可能引用格式不正确的UTF-8
/// 给这个结果一个生命期，就好像它是从_owner借用的一样
///
/// 安全：如果ptr不为空，它一定指向一个空字节结尾的可以安全访问的C字符串
unsafe fn char_ptr_to_str<T>(_owner: &T, ptr: *const c_char) -> Option<&str> {
    if ptr.is_null() {
        return None;
    } else {
        CStr::from_ptr(ptr).to_str().ok()
    }
}
```

这里永远不会用到 `_owner` 参数的值，但会用到它的生命期。把这个函数签名中的生命期写出来是这样的：

```
fn char_ptr_to_str<'o, T: 'o>(_owner: &'o T, ptr: *const c_char)
    -> Option<&'o str>
```

`CStr::from_ptr` 函数返回一个 `&CStr`，其生命期完全不受限制，因为它是从一个解引用的原始指针借用来的。不受限制的生命期几乎总是不确切的，因此应该尽快给它们添加限制。包含 `_owner` 参数导致 Rust 将它的生命期作为返回值类型的生命期，因此调用者可以接收到一个有更确切限制的引用。

根据 `libgit2` 的文档无法确定 `git_signature` 的 `email` 和 `author` 指针是否可以为空，这多少让人对 `libgit2` 还不错的文档有些失望。我们在源代码中四处翻查了一段时间，最终也没有找到让自己信服的证据。因此决定为防万一，最好让 `char_ptr_to_str` 做好面对空指针的准备。在 Rust 中，这类问题马上就能通过类型来回答：如果它是 `&str`，那你可以认为一定有字符串；如果它是 `Option<&str>`，那就是可选的。

最终，我们为需要的所有功能都提供了安全接口。而位于 src/main.rs 中的新 `main` 函数也明显“苗条了”许多，看起来像真正的 Rust 代码了：

```
fn main() {
    let path = std::env::args_os().skip(1).next()
        .expect("usage: git-toy PATH");
```

```
    let repo = git::Repository::open(&path)
        .expect("opening repository");

    let commit_oid = repo.reference_name_to_id("HEAD")
        .expect("looking up 'HEAD' reference");

    let commit = repo.find_commit(&commit_oid)
        .expect("looking up commit");

    let author = commit.author();
    println!("{} <{}>\n",
             author.name().unwrap_or("(none)"),
             author.email().unwrap_or("none"));

    println!("{}", commit.message().unwrap_or("(none)"));
}
```

本节在一个不安全 API 之上构建了一个安全 API，把违反前者协议的所有可能都转化为了一种 Rust 类型错误。最终结果是 Rust 可以保证你使用的接口的正确性。在很大程度上，我们让 Rust 强制执行的这些规则实际上也是 C 和 C++ 程序员最终会强制自己遵守的那些规则。Rust 之所以给人比 C 和 C++ 更严格的印象，不是因为这些规则有多么匪夷所思，而是因为这种强制是机械的和全面的。

21.9 小结

Rust 并不是一门简单的语言。它的目标是跨越两个截然不同的世界。它是一门现代编程语言，注重安全，也有类似闭包和迭代器这样的便捷特性。但 Rust 的目标是让你掌控运行它的机器的原始能力，而且运行时开销最低。

这门语言的形象就是由这些目标决定的。Rust 设法架起一座桥梁，以跨越不安全代码的鸿沟。它的借用检查器和零开销抽象，让开发者尽可能接近机器硬件同时又不会冒未定义行为的风险。如果这样还不够，或是如果你想利用已有的 C 代码，那不安全代码随时待命。但同样，这门语言并非只管提供这些不安全特性，然后说声祝你好运就完事了。它的目标仍然是使用不安全特性构建安全 API，也就是我们基于 `libgit2` 所做的。当然，还有 Rust 团队提供的 `Box`、`Vec`、其他集合类型，以及通道，等等。标准库中满满的安全抽象，在后台则是通过某些不安全代码实现的。

像 Rust 这样一门拥有雄心壮志的语言，恐怕注定不会是一套简单的工具。Rust 安全、快速、并发，而且高效。应该用它来构建大规模、高性能、安全、可靠的系统，以充分利用现有硬件的潜力。应该用它来让软件变得更好。

作者介绍

吉姆·布兰迪（Jim Blandy）从1981年开始编程，1990年开始编写自由软件。他曾是GNU Emacs和GNU Guile以及GNU调试器GDB的维护者。吉姆是Subversion版本控制系统最初的设计者之一。他目前在Mozilla开发Firefox的Web开发者工具。

贾森·奥伦多夫（Jason Orendorff）为Mozilla优化C++代码，是Firefox的JavaScript引擎的模块所有者。他是美国田纳西州那什维尔开发者社区的活跃成员，并会不定期组织当地的技术活动。

封面介绍

本书封面上的动物是一只蒙塔古蟹（Montagu's crab，Xantho hydrophilus）。这只蟹看起来很壮实，蟹壳有70毫米宽，给人感觉既结实又坚固。蟹壳的周边有沟槽状起伏，呈淡黄或红褐色。它有10条腿（5对），前面一对大腿（螯）大小相同，带有黑色尖爪或蟹钳，后面是3对粗壮而相对短小的步足，最后一对腿用来划水。它们走路和划水都是侧向的。

蒙塔古蟹发现于大西洋东北部和地中海地区。退潮时，它们生活在礁石或巨石下方。如果把石头抬起来，它们会攻击性地举起蟹钳并张开，以使自己看起来更大。

这种蟹以藻类、腹足类或其他种类的螃蟹为食。它们主要在夜间活动。每年3~7月是母蟹产卵季，幼蟹几乎整个夏季都浮游在海上。

O'Reilly图书封面上的很多动物是濒危生物，它们对这个世界很重要。要了解如何保护它们，请访问animals.oreilly.com。

本书封面图片取自*Wood's Natural History*。

O'REILLY®

Rust程序设计

Rust是一门新的系统编程语言，兼具C和C++的高性能和底层控制能力，而且能保证内存安全和线程安全，是系统编程发展史上的一个巨大进步。本书对Rust进行了全面介绍，详细解释了这门语言的重要概念，并提供了大量清晰易懂的示例，逐步指导读者用Rust编写出既安全又高性能的程序。

本书由两位具有数十年经验的系统程序员撰写，他们不仅分享了自己对Rust的深刻见解，而且还提供了一些建议和操作实践，对Rust开发者和系统程序员十分有帮助。

- **Rust如何在内存中表示值（辅以图表）**
- **完整解释了所有权、转移、借用和生命期**
- **Cargo、rustdoc、单元测试，以及如何在Rust公共包仓库上发布代码**
- **泛型代码、闭包、集合和迭代器等高级特性**
- **Rust中的并发：线程、互斥量、通道和原子操作**
- **不安全代码，以及如何保持使用常规代码的完整性**
- **用丰富的例子展示了Rust各方面特性的综合运用**

“本书是Rust开发者案头必不可少的新书。它能教会你正确的学习方法，尤其是掌握所有权和生命期的概念。书中满满的实战示例和对真实问题的深度思考让我印象深刻。”

——Raph Levien
Google

吉姆·布兰迪（Jim Blandy），Mozilla软件工程师，拥有近40年编程经验和30年自由软件开发经验，是Subversion版本控制系统最初的设计者之一。

贾森·奥伦多夫（Jason Orendorff），资深软件工程师，拥有20余年软件开发经验，目前在为Mozilla Firefox Web浏览器开发JavaScript引擎。

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议　计算机/程序设计/Rust

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



扫码加入
后端交流群

ISBN 978-7-115-54649-4



定价：139.00元